Section 10
Aircraft Situation Display Function

Purpose

The *Aircraft Situation Display* (*ASD*) is an application used by the traffic management specialist to access and display various ETMS data. The *ASD* displays ETMS data in a variety of ways including graphical displays superimposed on map overlays; textual reports of flight and traffic activity; and bar graphs of traffic demand counts. The *ASD* is interactive and responds to traffic management specialist requests as entered through use of the keyboard and mouse (or trackball). At least one *ASD* runs on each traffic management specialist workstation.

> **NOTE:** It is recommended that the reader become thoroughly familiar with the operation of the *ASD* before reading this section. The operation of the *ASD* is described in the *ASD Tutorial* and the *ASD Reference Manual.*

Execution Control

The execution of the *ASD* is initiated from a script. The script may be a manually invoked shell script, a *login* script, or more typically, a script invoked by clicking on the *Tool Manager's* ASD icon. The setup parameters (see the next Input section), which are read by the *ASD* on startup, are used to set up inter-process communications with the appropriate network *site* as well as determining the behavior of the *ASD.*

Input

The *ASD* gets the following dynamic data from other ETMS processes and displays it to the traffic management specialist:

- *ASD* updates — **map** and **rte** files received from the *Flight Table Manager*, which are used to update the flight position/flight data displays. Files are received periodically based on an *FTM* parameter. Typically, files are received every one minute.
- Flight data replies — responses from the *Flight Table Manager* to requests for detailed flight information. Used by the *ASD* to draw alerted flight paths.
- Alert updates — files containing alerted elements, and the traffic demands at each alerted element. Used by the *ASD* to update the alerted element display and to draw bar charts for alerted elements.
- Flight list, flight count, ARRD, and alert reports — reports that contain flight

and traffic data requested by the traffic management specialist. Received on request from the *Listserver*.

- SA and FT Reports — weather reports that contain surface observation and ter-minal forecast data, respectively. Received on request from the *Weather Server*.
- Demand Data Replies — traffic demand data received on request from the *Traffic Demands Database* or from the *Alert Server Process*, and used to draw time bar and bar chart data.
- Capacities — data received on request from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.
- GA Estimates — data received on request from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.

The *ASD* reads the following *startup* files when it is initially invoked:

- **setup** — startup configuration parameters such as primary and secondary *site* designations and whether to keep an error log.
- **adapt_default** — user-adapted *ASD* commands that are automatically executed upon startup.
- **colors_default** — user-adapted colors for drawing the *ASD* overlays.
- **audible_alarm** — definitions of alerted elements colors, duration of an individual beep tone, number of beeps per cycle in one second, number of ringing cycles, and a list of alerted element names.
- **remarks_keywords** – user-adapted search criteria for use when searching for flights that fall under particular keywords.

The *ASD* reads the following optional arguments from the command line that initiates program execution:

a — specify the name of the **a**daptation script file.

b — start the *ASD* without a standard window frame but with a white **b**order drawn around the window's circumference.

d — activate the **d**ebug mode and do not create a full screen window for use with the debugger.

e — use **e**xhibit mode for Smithsonian exhibit.

i — use **i**nternational mode to project the geographical data when displaying map or aircraft.

k — specify the **k**eyboard test repeat count.

l — use **l**arge font display.

m — simulate a **m**onochrome node on the color one.

n — indicate **n**o reply time out for each request made.

s — operate in **s**tand—alone mode with no connection to the network addressing message switching system.

t — set flight update **t**ime out in minutes.

w — set the **w**ait time between keyboard tests in one second.

x — use e**x**perimental mode for testing new map databases.

% — specify QA mode for testing the canned alert data file.

The *ASD* reads data from the following static data files:

- **map.gpr.5** — names and locations/boundaries of all airports, NAVAIDs, jet airways, Victor airways, ARTCCs, sectors, arrival fixes, departure fixes, and SUA's. Used to draw map overlays and alerts.
- **airway.db.5**, **airway.index.5** — detailed data for jet and Victor airways and the indexes for searching for specific airways. Used to display a single airway requested by a traffic management specialist.
- **runway.db.5**, **runway1.index.5**, **runway2.index.5** — runway locations for all airports and the indexes for searching for a specific airports runway layout.
- **fix.font** — font used to label arrival and departure fixes.
- **sua.font** — font used to label all types of Special Use Areas (SUAs).
- **time_bar.font** — font used to label the monitor/alert time bar.
- **artcc.font** — font used to label the ARTCCs.
- **sector.font** — font used to label all types of sectors.
- **route.font** — font used to label jet and Victor airways.
- **airport.font** — font used to label airports when all airports are displayed.
- **pacing_airport.font** — font used to label airports for the pacing airport display.
- **navaid.font** — font used to label NAVAIDs.

- **data_block.font** —font used for flight data blocks.
- **airplane.font** — symbols for drawing airplanes.
- **report.font** — font used to display reports.
- **bold.font** — bold character font used for menu entries and prompts.
- **font24** —font 24 pixels wide.

- **weather_symbols.24** — weather symbols used for small windows.
- **weather_symbols.36** — weather symbols used for large windows.
- **etms_icons.1** —set of window icons tailored for each ETMS function.

The *ASD* optionally reads data from the following types of files in response to user commands:

- Colors files — color setting for each overlay adapted by using the **adjust colors** command and stored with the **save colors** command.
- Script files — *ASD* scripts created by the user.
- Weather files — hand-drawn weather patterns saved by the **write weather** command.
- WX_maps files — weather products from the weather server.

Output

The *ASD* sends the following dynamic data in the form of messages to other ETMS processes:

- Flight data requests — requests for detailed flight information from the *Flight Table Manager*. Used by the *ASD* to draw alerted flight paths.
- Flight list, flight count, ARRD, and alert report requests — requests for reports as entered by the traffic management specialist.
- SA and FT report requests — requests for weather reports that contain surface observation and terminal forecast data, respectively, from the *Weather Server*.
- Demand Data Requests — requests for traffic demand data from the *Traffic Demands Database* process and *Alert Server Process* are used to request time bar data, bar chart data, and alerted flight data.
- Capacity queries — requests for capacity data from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.
- Capacity updates — capacity data entered by the traffic management specialist to update the traffic demands database.
- General Aviation (GA) estimate queries — requests for GA estimate data from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.
- General Aviation (GA) estimate updates — GA estimate data entered by the traffic management specialist to update the traffic demands database.

- Schedule database updates — updates to the schedule database entered by the traffic management specialist; consist of **FPSD**, **CXSD**, **INHB**, and **ACTV** commands.

The *ASD* optionally writes data to the following files in response to user commands:

- **Colors** file — color setting for each overlay adapted by using the **adjust colors** command and stored with the **save colors** command.
- **Weather** file — hand-drawn weather patterns stored with the **write weather** command.

The *ASD* optionally writes the trace data to the following file (if the **setup** file *log errors* directive):

- Trace file — timing data and a log of *ASD* commands, error messages, and execution traceback.

Design Issue: Graphics Support Software

The *ASD* uses the HP/Apollo *Graphics Primitives (GPR)* package to do all graphics, keyboard input, and mouse input. The *GPR* package is the lowest—level graphics support software provided by HP/Apollo. *GPR* was chosen, because it allows for customizing the use of the graphics hardware for optimal performance. *GPR* is fully integrated with the Apollo operating system and display manager software.

GPR is used in *direct* mode, i.e., the program operates within a display manager window; it does not borrow the display. The main reasons for using direct mode is to be able to use the HP/Apollo debugger to debug the program and to allow other Apollo shells/applications to co—exist with the *ASD*.

The processing of the *ASD* is heavily dependent on the *GPR* routines. The reader should refer to the HP/Apollo *GPR* documentation as needed.

Design Issue: Constructing Addresses

The *ASD* communicates with other processes via the network addressing message switching system. At startup, the *ASD* connects to the *node switch*, and constructs the addresses of the processes with which it needs to communicate by combining site names, class names, and wildcards for node, invocation number, and subaddress.

The *ASD* constructs and stores wildcard addresses rather than requesting explicit addresses from the message switching system. Constructing the addresses eliminates the need for request/reply between the ASD and the network addressing system that would be necessary to acquire explicit addresses. This reduces *ASD* startup time.

Also, wildcarding the addresses allow the ASD to be less sensitive to other processes being restarted. A message addressed with wildcards will successfully make it to its destination

regardless of the node on which the destination process is running or the invocation number of the process.

Design Issue: Site Coordination

In order to support data consistency and system integrity, a field site's processes should receive data from a single hubsite processing string. Coordination of data sources among field site processes is controlled by the source site of flight data for the field site's FTM.

The FTM supplies the ASD with the current source site as part of each map file update. The ASD uses this information to select which set of constructed addresses to consider current. The ASD uses the current set of addresses when making requests for data.

For list requests, the ASD supplies the *Listserver* with an integer value in the subaddress field of the *Listserver* address. This integer represents the site to which the *Listserver* should direct the request. The ASD and *Listserver* share a configuration file (*asd_list_site_data*) to map integer values to site names. In this manner, the FTM, ASD, and *Listserver* all receive data from the same hubsite processing string.

Processing Overview

Logically, the processing performed by the *ASD* can be grouped into three main modules: *Initialize*, *Process Input*, and *Draw Displays*. Data flow for these items is shown in Figure 10-1.

The *Initialize* module sets up many display parameters, such as display type, window size, display time-out, and font styles (using the various **fonts** files). The *Initialize* module displays a title page to the user and draws the initial *ASD* display.

The *Initialize* module also responds to the following conditions:

- If the **setup** file exists, it reads in the startup configuration parameters.
- If an **adapt_default** file exists, it reads this file and performs the adaptation commands.
- If a **colors_default** file exists, it reads this file and sets up the display color map for each overlay.
- If an **audible_alarm** file exists, it reads this file and sets up the audible alarm for use in monitoring the alerted elements.

The *Initialize* module checks for the existence of the data files required for later processing. The *Initialize* module also establishes a connection to the *node switch* to support communication with other ETMS Functions.

The *Process Input* module is the driver for the operation of the *ASD*. This module repeatedly checks for keystrokes, mouse button clicks, and cursor movements. When a user command is entered, the *Process Input* module responds to the command. If the command requires data from an external source, it sends a data request.

The *Process Input* module also does the following:

- For some requests, waits for the reply before continuing. For other requests, continues with other processing after the request is issued.
- Checks repeatedly for asynchronous network messages sent from other ETMS functions. When a network message is received, initiates the display of the received data.
- Reads and write **colors** and **defaults** files in response to user commands.
- Reads **script** and **weather** files to respond to user commands.
- Uses location names from the **map.gpr.5** file to help interpret and verify the geographical names or airways as specified in the user commands.

The *Draw Displays* module consists of a set of routines which execute the various data displays. The *Draw Display* routines are invoked by the *Initialize* and *Process Input* modules as needed to perform their functions. The *Draw Display* routines use data from the **map.gpr.5**, **runways**, and **airways** files to generate overlay graphic displays.

The *Initialize*, *Process Input*, and *Draw Displays* modules communicate with each other through a vast array of global variables. Global variables were used, because many routines are dependent on common parameters, and because the execution thread of the *ASD* program can be extremely varied.

For example, a set of global flags, *displayed*, are used to indicate which of the overlays are currently displayed. When the user *toggles* an overlay of the displayed map data through a keyboard command (e.g., weather patterns or flights), the *Process Input* module toggles the corresponding flag in *displayed*. When the display is re-drawn, the *Draw Displays* routines draw or erase the corresponding overlay data display.

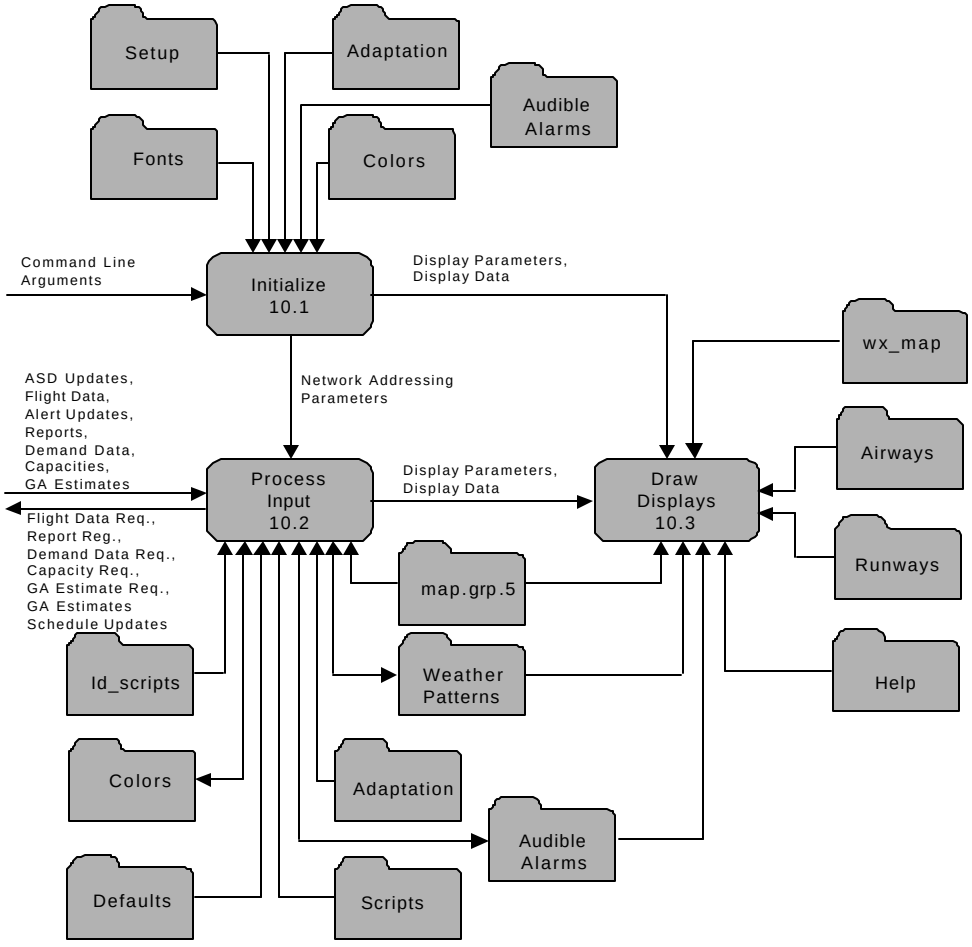

Figure 10-1. Data Flow of the Aircraft Situation Display

For simplicity, the *ASD* is assumed to be running with full network support. The *defaults* file folder is used generically for any defaults data files.

10.1 The Initialize Module

This module performs the initialization for the *ASD* function.

Input

The *Initialize* module uses the following data received dynamically from the *Flight Table Manager*:

- ASD updates — **map** and **rte** files received from the *Flight Table Manager* used to draw the initial flight position/flight data display. The *Flight Table Manager* sends the latest **map** and **rte** files when an *ASD* first registers.

The *Initialize* module reads the following startup files when it is initially invoked:

- **setup** — startup configuration parameters such as primary and secondary *site* designations and whether to keep the error log.
- **adapt_default** — user-adapted *ASD* commands that are executed upon startup.
- **colors_default** — user-adapted colors for drawing the *ASD* overlays.
- **audible_alarm** — definitions of alerted elements colors, duration of an individual beep tone, number of beeps per cycle in one second, number of ringing cycles, and a list of alerted element names.

The *Initialize* module reads the following optional arguments from the command line which initiates the program execution:

a — specify the name of the **a**daptation script file.

b — start the *ASD* without a standard window frame but with a white **b**order drawn around the window's circumference.

d — activate the **d**ebug mode and do not create a full screen window for use with the debugger.

e — use **e**xhibit mode for Smithsonian exhibit.

i — use **i**nternational mode to project the geographical data when displaying map or aircraft.

k — specify the **k**eyboard test repeat count.

l — use **l**arge font display.

m — simulate a **m**onochrome node on the color one.

n — indicate **n**o reply time out for each request made.

s — operate in **s**tand-alone mode with no connection to the network addressing message switching system.

t — set flight update **t**ime out in minutes.

w — set the **w**ait time between keyboard tests in one second.

x — use e**x**perimental mode for testing new map databases.

% — specify QA mode for testing the canned alert data file.

The *Initialize* module reads data from the following static data files:

- **map.gpr.5** — contains names and locations/boundaries of all airports, NAVAIDs, jet airways, Victor airways, ARTCCs, sectors, arrival fixes, departure fixes, and SUA's. Used to draw map overlays and alerts.
- **fix.font** — font used to label arrival and departure fixes.
- **sua.font** — font used to label all types of SUA.
- **time_bar.font** — font used to label the monitor/alert time bar.
- **artcc.font** — font used to label the ARTCCs.
- **sector.font** — font used to label all types of sectors.
- **route.font** — font used to label jet and Victor airways.
- **airport.font** — font used to label airports when all airports are displayed.
- **pacing_airport.font** — font used to label airports for the pacing airport display.
- **navaid.font** — font used to label NAVAIDs.
- **data_block.font** — font used for flight data blocks.
- **airplane.font** — symbols for drawing airplanes.
- **report.font** — font used to display reports.
- **bold.font** — bold character font used for menu entries and prompts.
- **font24** — font 24 pixels wide.
- **weather_symbols.24** — weather symbols used for small windows.

- **weather_symbols.36** — weather symbols used for large windows.
- **etms_icons.1** — set of window icons tailored for each ETMS function.

Output

The *Initialize* module outputs the following data through global variables:

- Display parameters — many variables including display type (color/bw), number of display color planes, display color set, current window size, maximum window size, font index array, color array, fill patterns, menu box sizes, cursor position, data displayed flags, map center, translation offsets, zoom scale, audible alarm settings, error log flag, and special mode flags (for experimental mode, large screen mode, etc.).
- Display data — static and dynamic data passed to the *Draw Display* routines as needed to draw the initial display overlays (depends on the adaptation file contents). By default, consists of an *ASD* update.
- Window icon — window icon character changed to the one tailored for the *ASD* function.

Processing

The *Initialization* module processing consists of a long series of steps which are executed sequentially from the main program module. Processing is performed by invoking separate routines, which appear in parentheses in the description that follows.

The *Initialization* module proceeds in the following sequence:

(1) Read the command line arguments and set a corresponding flag for each argument that exists. For instance, if a flight time out argument exists, set **flight_update_timeout** to the specified value. (*check_args*)

(2) Read the startup parameters from the **setup** file. (*read_setup_file*)

(3) Set **flight_update_timeout** to 7 minutes as a default if no command line argument **—t** specified.

(4) Create /reports, /rawlist, /etms5/asd/adapt, /etms5/asd/adapt/weather, and **/etms5/asd/adapt/scripts** directories if they do not exist.

(5) Set *GPR* mode to direct.

(6) Set display parameters depending on special modes, display type, and current window size: x and y bit ranges, maximum window size, small window size, number of display color planes, and color flag. Bring window to front of display if obscured.

(7) Read **font** files and load the fonts into GPR. If **large_screen** is set, read the font files in the **large_fonts** directory; otherwise, read the files in the **fonts** directory Create the **font_id** array of GPR font names indexed by the display element types (arrival_fixes, sectors, etc.). (*set_up_fonts*)

(8) Set **font_id** array values that are not defined explicitly in the fonts files by making them equivalent to values that are.

(9) Set menu box sizing parameters appropriate for the **bold_fonts** character size. (*adjust_panel_size*)

(10) Set the window parameters based on the current window size. (*measure_window*)

(11) Display the *ASD* title page. (*display_title*)

(12) Allocate two *GPR* bitmaps, and fill with the background colors.

(13) Create and initialize the cursor position and attributes.

(14) Create striped fill patterns, which are used later to draw sector alerts. (*stripes)*

(15) Load the map overlay data from the **map.gpr** file.(*mapl)*

(16) Initialize the zoom scale, map center point, and *displayed* flags to hard—coded default values. Compute the translation offsets. Read the **audible_alarm** file to initialize the alarm settings if it exists. (*initialize*)

(17) Read the **colors_default** file to initialize the color arrays *color_table* if the color file exists; otherwise, set the color array to the default values. Each entry in the array specifies the color for the corresponding overlay. Separate color variables exist for controlling the menus, prompts, time bars, and bar charts. If **color_display** is true, the colors are set to a hard—coded set (*setup_color_node*); otherwise, all foregrounds are set to black and all backgrounds are set to white (*setup_bw_node*).

(18) Check for the existence of the **airway** and **runway** static data files. (*check_for_files*)

(19) Connect to the *nodeswitch* process. Register with the FTM for periodic **map** and **rte** updates. (*asd_init_nwa*)

(20) Execute the **adapt_default** file if exists. (*adapt*)

(21) Draw the display background and map overlays. (*view*)

(22) If the *flights* flag is not set in **displayed**, ask for the correct time from the *FTM* process. (*flight_check*)

Error Conditions and Handling

Errors incurred during the *Initialize* module can be fatal or non-fatal. Non-fatal errors cause an error message to be displayed, but the *ASD* continues to execute. Fatal errors cause the *ASD* to terminate execution.

The following **non-fatal** error may occur during the *Initialize* processing:

- Cannot find or open an **airway** or **runway** data file (*check_one_file*).

The following **fatal** errors may occur during the *Initialize* processing:

- Cannot open or read **setup** file (*read_setup_file*).
- Cannot load a *GPR* font (*load_font*).
- Cannot load map overlays into *GPR* (*mapl*).
- Incompatible version of the **map.gpr** file (*mapl*).

Before the *ASD* terminates its execution, it *cleans up* by executing the following steps (in *cleanup_handler* procedure):

(1) Close all open report windows (*close_all_windows*).

(2) Terminate GPR. (*gpr_$terminate*)

(3) Disconnect from the FTM. (asd_register_for_services)

(4) Disconnect from the *ASP*. (*asd_register_for_services*)

(5) Delete all *pad.?** files in the trace directory. *(dlf)*

(6) Restore the window pad to normal. *(pad_$cooked)*

(7) Reset the current working directory. *(name_$set_wdir)*

(8) Exit the ASD. (pgm_$exit)

10.2 The Process Input Module

The *Process Input* module is the main driver for the *ASD* function. This module accepts user commands entered through the keyboard/mouse and data input from other ETMS functions via network messages, then takes the appropriate action. The *Process Input* module invokes routines from *Draw Displays* to generate outputs on the screen.

Input

The *Process Input* module gets the following dynamic data from other ETMS functions for display to the traffic management specialist:

- ASD updates — **map** and **rte** files received from the *Flight Table Manager*, which are used to update the flight position/flight data displays.
- Flight data replies — responses from the *Flight Table Manager* to requests for detailed flight information. Used to draw alerted flight paths.
- Alert updates — files containing the alerted elements, and the traffic demands at each alerted element. Used to update the alerted element display and to draw bar charts for alerted elements.
- Flight list, flight count, ARRD, and alert reports — reports that contain flight and traffic data requested by the traffic management specialist. Received on request from the *Request Server*.
- SA and FT reports — weather reports that contain surface observation and terminal forecast data, respectively. Received on request from the *Weather Server*.
- Demand Data replies — traffic demand data received on request from the *Traffic Demands Database* or from the *Alert Server Process* is used to draw time bar and bar chart data.
- Capacities — data received on request from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.
- GA estimates — data received on request from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.

The *Process Input* module reads data from the following static data files:

- **map.gpr.5** — names and locations/boundaries of all airports, NAVAIDs, jet airways, Victor airways, ARTCCs, sectors, arrival fixes, departure fixes, and SUA's. Used to interpret and verify user inputs.
- **airway.db.5**, **airway.index.5** — detailed data for jet and Victor airways and the indexes for searching for specific airways. Used to look up a single airway

requested by a traffic management specialist.

- **runway.db.5**, **runway1.index.5**, **runway2.index.5** — runway locations for all airports and the indexes for searching for a specific airports runway layout.

The *Process Input* module optionally reads data from the following types of files in response to user commands:

- **adapt_default** — user-adapted *ASD* commands that are automatically executed upon startup.
- Colors files — color setting for each overlay adapted by using the **adjust colors** command and stored with the **save colors** command.
- Script Files — *ASD* scripts created by the user.
- Weather files — hand-drawn weather patterns saved by the **write weather** command.
- WX_maps files — weather products from the weather server.

The *Process Input* module gets the following data from the *Initialize* module through global variables:

- Display parameters — many variables including display type (color/bw), number of display color planes, display color set, current window size, maximum window size, font index array, color array, fill patterns, menu box sizes, cursor position, data displayed flags, map center, translation offsets, zoom scale, audible alarm settings, error log flag, and special mode flags (for experimental mode, large screen mode, etc.).

Output

The *Process Input* module sends the following dynamic data to other ETMS functions:

- Flight data requests — requests for detailed flight information from the *Flight Table Manager*. Used to draw alerted flight paths.
- Flight list, flight count, ARRD, and alert report requests — requests for reports as entered by the traffic management specialist.
- SA and FT report requests — requests for weather reports containing surface observation and terminal forecast data, respectively, from the *Weather Server*.
- Demand Data requests — requests for traffic demand data from the *Traffic Demands Database Processor*. Used to draw time bar data.
- Capacity requests — requests for capacity data from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.

- Capacities — capacity data entered by the traffic management specialist to update the traffic demands database.
- GA estimate requests — requests for GA estimate data from the *Traffic Demands Database Processor.* Used to respond to traffic management specialist requests.
- GA estimates — GA estimate data entered by the traffic management specialist to update the traffic demands database.
- Schedule database updates — updates to the schedule database entered by the traffic management specialist; consist of **FPSD**, **CXSD**, **INHB**, and **ACTV** commands.

The *Process Input* module optionally writes data to the following files in response to user commands:

- **Colors files** — color setting for each overlay adapted by using the **adjust colors** command and stored with the **save colors** command.
- **Weather files** — hand-drawn weather patterns saved by the **write weather** command.

The *Process Input* module sends the following data to the *Draw Display* routines:

- Display parameters — many variables including display type (color/bw), number of display color planes, display color set, current window size, maximum window size, font index array, color array, fill patterns, menu box sizes, cursor position, data displayed flags, map center, translation offsets, zoom scale, audible alarm settings, error log flag, and special mode flags (for experimental mode, large screen mode, etc.).
- Display data — dynamic data needed to draw the many displays that can be requested by the user. Includes *ASD* updates (active flight data), flight list and count reports, ARRD reports, weather reports, alerts, bar charts, alert reports, and alerted flight displays.

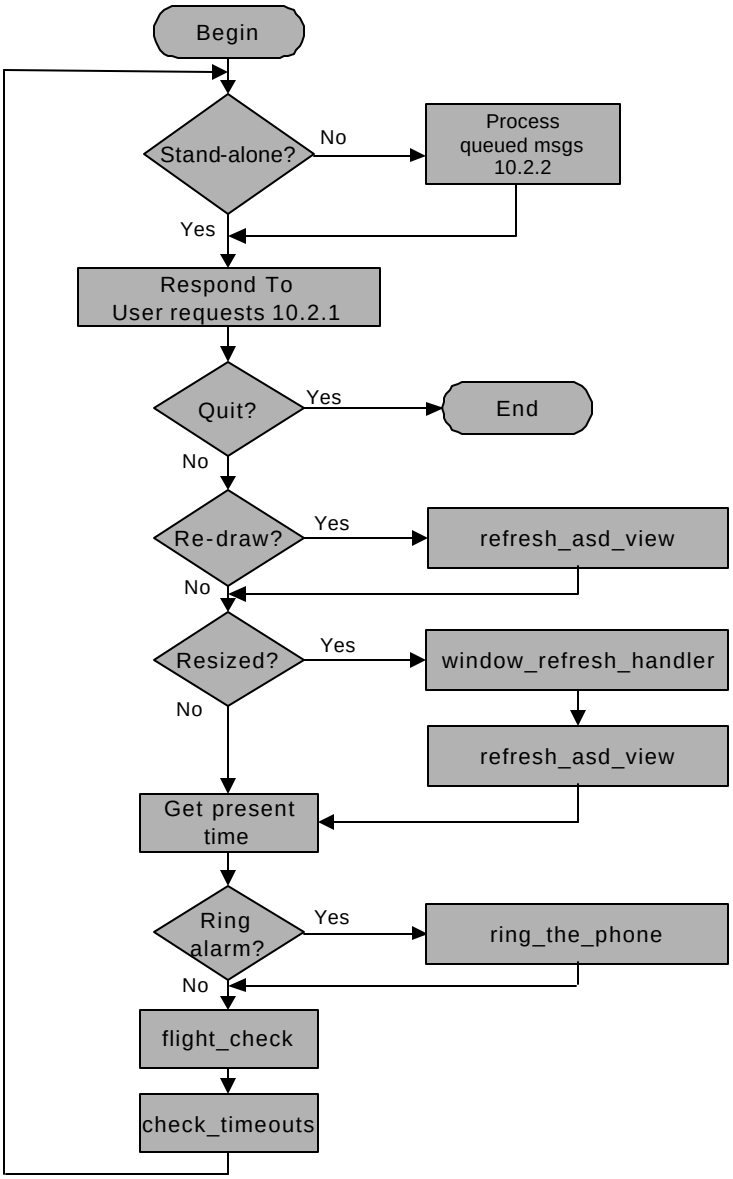

Figure 10-2. Main Logic for the Process Input Module

Processing

The main logic of the *Process Input* module is shown in Figure 10-2. The *Process Input* module starts looping after the *Initialize* module has completed the initialization processing and continues looping until either a fatal error occurs (not shown) or the user enters a **quit** command.

On each pass of the *Process Input* module, the *Process Queued Messages* module performs the following sequence.

(1) If the *ASD* is not started up as a stand—alone process, the module checks for incoming network messages from other ETMS functions.

(2) If the received message is a status reply of the previous user request, it calls the *Draw Displays* module (Section 10.3) to display the status message.

(3) If the received message is a map update from the *FTM* process and flights is set in the *displayed* variable, it calls the Draw Displays module to redraw the *ASD* window to show new traffic data.

(4) If the received message is from the *ASP* process, it calls the *Draw Displays* module to redraw the alerted elements, time bar, and bar chart (if displayed).

On each pass of the *Process Input* module, the *Respond To User Requests* module performs the following functions:

- Checks for any user input. The user enters requests in three ways: typing single keystrokes, entering semicolon commands, or using menus. If any input is present, the *Respond To User Requests* module obtains whatever information it needs from the user and invokes a routine to execute the request.

- Detects if the *ASD* window needs to be refreshed. If TRUE, calls the *refresh_asd_view* routine to refresh the window. If the *ASD* window was resized, invokes the *window_refresh_handler* routine to enable refresh condition and then invokes the *refresh_asd_view* to update the screen. The *refresh_asd_view* routine is part of the *Draw Displays* module.

- Examines the current time to see if the alerts alarm should go off. If so, the *ring_the_phone* routine is called to flash the alerted elements and ring the alarm.

- Gets the correct time from the *FTM* process and checks if any of the user requests have expired without a reply. Whenever a data request is made of any external function, the requesting routine sets a bit in the global variable **wait_flags** and puts a wait time into the global array **wait_times**. If any **wait_flags** are set, the *check_time outs* routine checks the corresponding value in **wait_times** against the current time.

On each pass of the *Process Input* module, *check_timeouts* is invoked to perform the time out checks. If a user request has expired before the *Process Input* module receives any response, the corresponding error message will be displayed. Then, the request is removed from the waiting queue and a flag is reset to indicate that the request is no longer outstanding.

10.2.1 The *Respond To User Requests Module*

The *Respond To User Requests* module accepts three general classes of input from the user: keyboard commands, semicolon commands, and menu commands. Keyboard commands are single keystrokes. Semicolon commands are strings of text consisting of a semicolon followed by the command name and a variable number of fields. Menu commands are entered interactively using the mouse buttons. When the middle or right mouse button is depressed, a menu is displayed. The user moves up and down the menu entries by moving the mouse. The user can move to sub—menus by sliding the cursor off to the right. A menu selection is made by releasing the mouse button when the cursor is in the desired entry. The logic for processing a user request is shown in Figure 10-3.

The HP/Apollo Display Manager allows a user to type ahead when the node is busy. Therefore, when the *Respond To User Requests* module asks for input (through a *GPR* routine call), it may get one request or several requests. The *Respond To User Requests* module processes all requests before returning to the *Process Input* module.

The *Respond To User Requests* module first checks whether a mouse button has been pressed. There are four different conditions under which a mouse button may be interpreted:

(1) If the user is drawing an experimental route (indicated by the status of the **drawing_experimental_route** flag), pressing a mouse button causes either *seek_navaid*, *display_lat_lon*, or *close_route* to be invoked.

(2) If the user has placed the cursor in a time bar interval, a time range selection message is displayed and *Process Input* module then waits for the second click on the time bar to end the time range selection.

(3) If the user has placed the cursor in a report name icon, pressing a mouse button causes either *create_window* (to display the report), *print_report*, or *delete_selected_icon* to be invoked.

(4) If none of these three conditions applies, the *Process Input* module checks whether (a) the mouse input was a single click or (b) a click/drag combination.

In case (a), the data block of the nearest flight is displayed if the left mouse button was clicked. If the middle mouse button was clicked, the *Do Menu* module (Section 10.2.1.1) is called to display the main menu. If the right mouse button was clicked, the *Do Menu* module is called to display another menu for the selected object (e.g. data block or flight icon).

In case (b) (click/drag combination), the data block of the nearest flight is displayed or repositioned if there is at least a flight icon displayed.

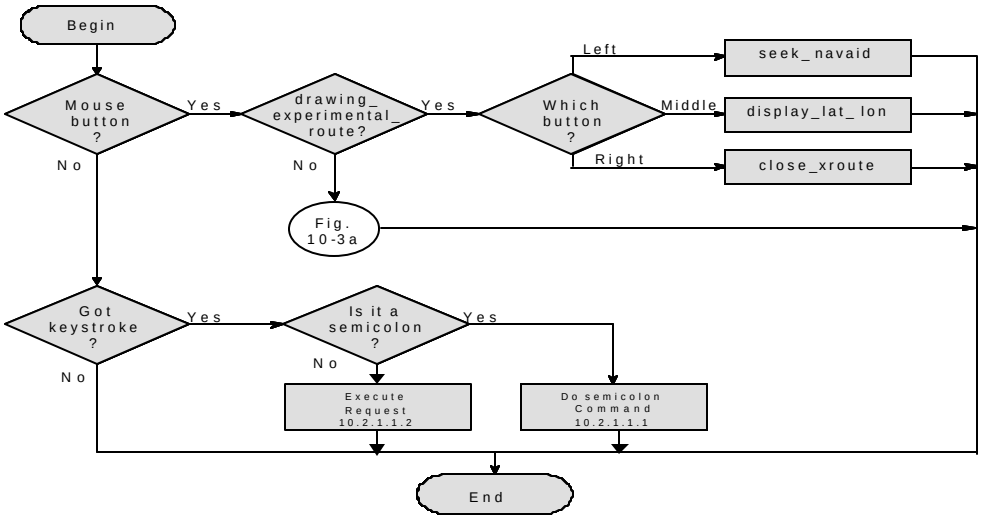

Figure 10-3. Logic for the Respond to User Requests Module

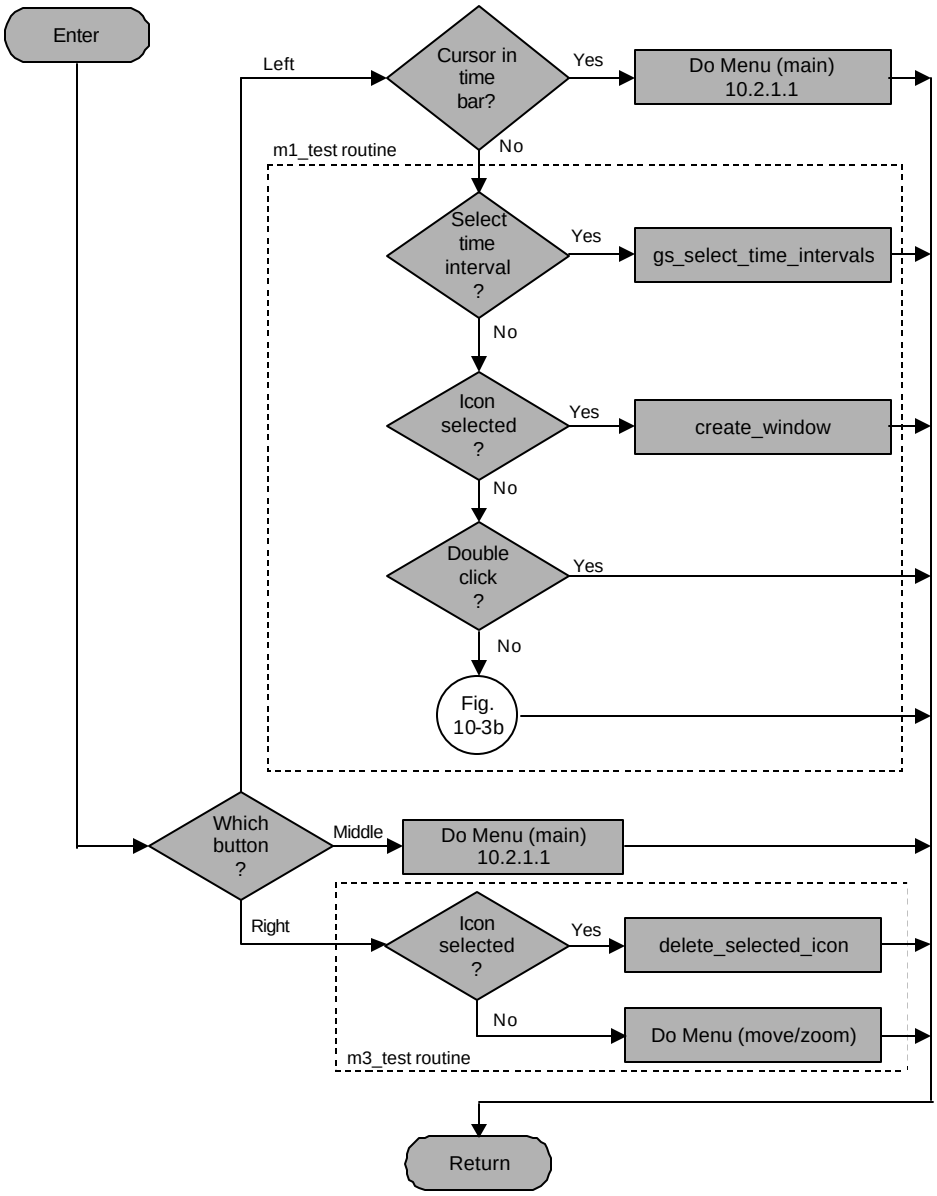

Figure 10-3a. Logic for the Respond to User Requests Module (continued)

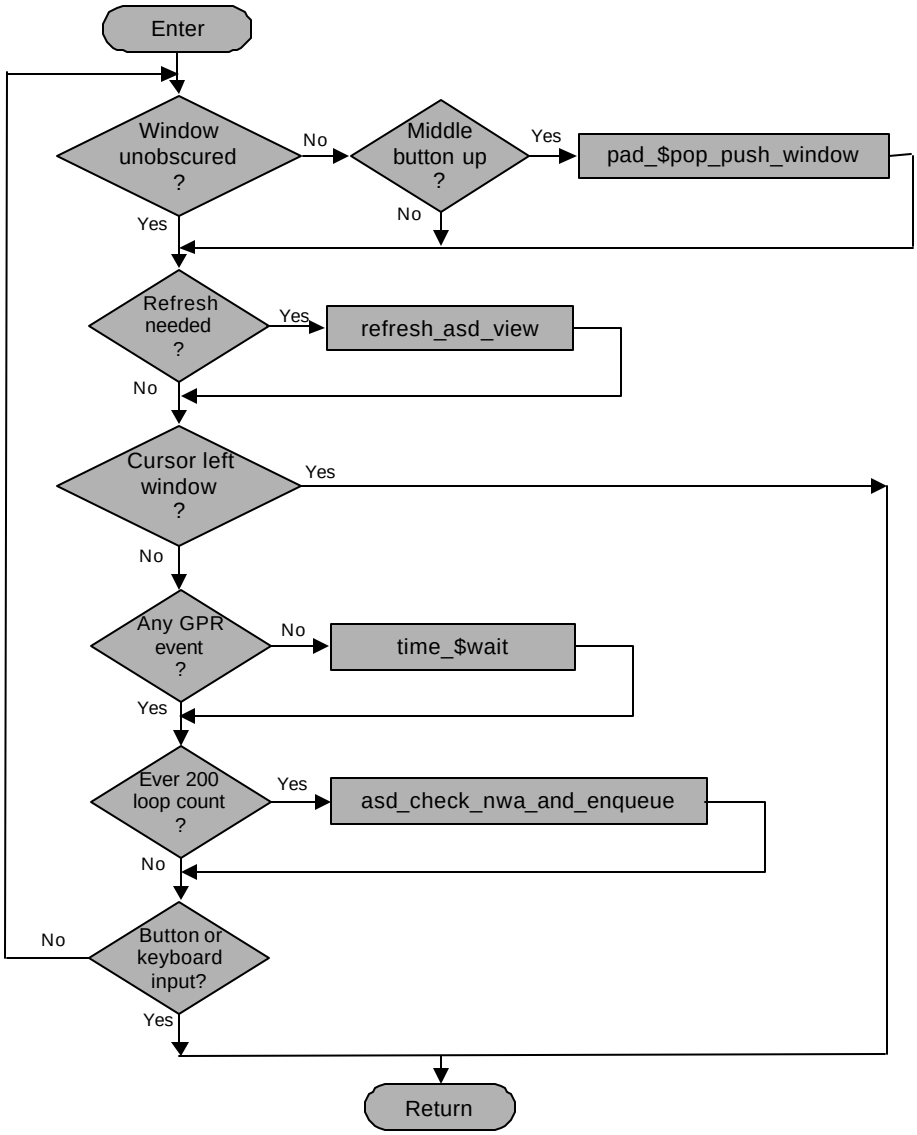

Figure 10-3b. Logic for the Respond to User Requests Module (continued)

If a mouse button has not been pressed, the *Respond To User Requests* module checks for a keystroke. If the semicolon has been pressed, the *Do Semicolon Command* module is invoked. If not, the keystroke is a keyboard command; the *Respond To User Requests* module invokes a routine from the *Execute Request* module depending on the key that was pressed.

Regardless of the method used for entering the command, the *Respond To User Requests* module eventually invokes a routine from the *Execute Request* module to take the appropriate action. The routine is invoked directly if the entry is a keyboard command; otherwise it is invoked from the *Do Semicolon Command* module or from the *Do Menu* module.

If arguments are needed to perform the request, the *Execute Request* routine prompts the user for the data. When the request requires a potentially lengthy response time from another ETMS function, the *Execute Request* routine initiates the request and allows the *ASD* processing to continue.

Later, the *Process Queued Messages* module completes the request when the data reply is received. The following sections describe the *Do Menu* module (Section 10.2.1.1), the *Do Semicolon Command* module (Section 10.2.1.1.1), and the *Execute Request* module (Section 10.2.1.1.2). The *Process Queued Messages* module is described in Section 10.2.2.

10.2.1.1 The *Do Menu* Module

The *Do Menu* module consists of two types of routines: generic menu handling utilities and routines which draw the specific menus. Each menu that appears on the display has corresponding routines to handle menu entries and to take the appropriate action for the user input.

When a user slides off one menu to another menu, the first menu routine invokes another menu routine. Each menu routine uses a constant **level**, which defines where the menu belongs in the menu hierarchy. The menu routines invoke the menu utilities for such functions as drawing the menus and checking the cursor position.

The menu routines and utilities communicate through several global arrays, which determine the menu status as follows:

- **cross_hatch** indicates whether any of the menu boxes should be drawn with cross-hatches.
- **selection** indicates which entry is currently selected

  selection can also be set to the values **advanced** (meaning the cursor has been moved off to the right), **retreat** (indicating that the cursor has been moved back to the left), and **give_up** (indicating that the cursor has been moved completely off the menus).
- **previous_selection** saves the entry that was selected when the cursor was

moved off a menu.

A menu routine executes in the following sequence:

(1) determines whether any of its menu entries are active (e.g., are the high sectors turned on?) by examining global flags and if so, adds that entry to the **cross_hatch** set.

(2) calls *pop_menu* to draw the menu at the current cursor position.

(3) enters a loop that continually checks the cursor position and mouse button activity until a valid selection is made.

The user *retreats* to the next higher level, or the user *gives up*. If a valid selection is made, the appropriate routine from the *Execute Request* module is invoked, the menus are erased, and the routine exits. If the user retreats to a higher level, the lowest level menu is erased, and the menu routine returns. If the user gives up, all the menus are erased, and the routine returns.

Multi-level menus are handled recursively. When the user *advances* to a lower-level menu, the menu routine for the lower-level menu is invoked within the loop of the higher-level menu. The lower level menu must then be resolved in the same manner by selecting, retreating, or giving up.

When the lower-level menu returns, the higher-level menu may still be active or may be resolved by the action taken at the lower level. The menu routines can theoretically operate to any number of levels; however, the current deepest level is four menus.

The logic of a generic menu routine is shown in Figure 10-4.

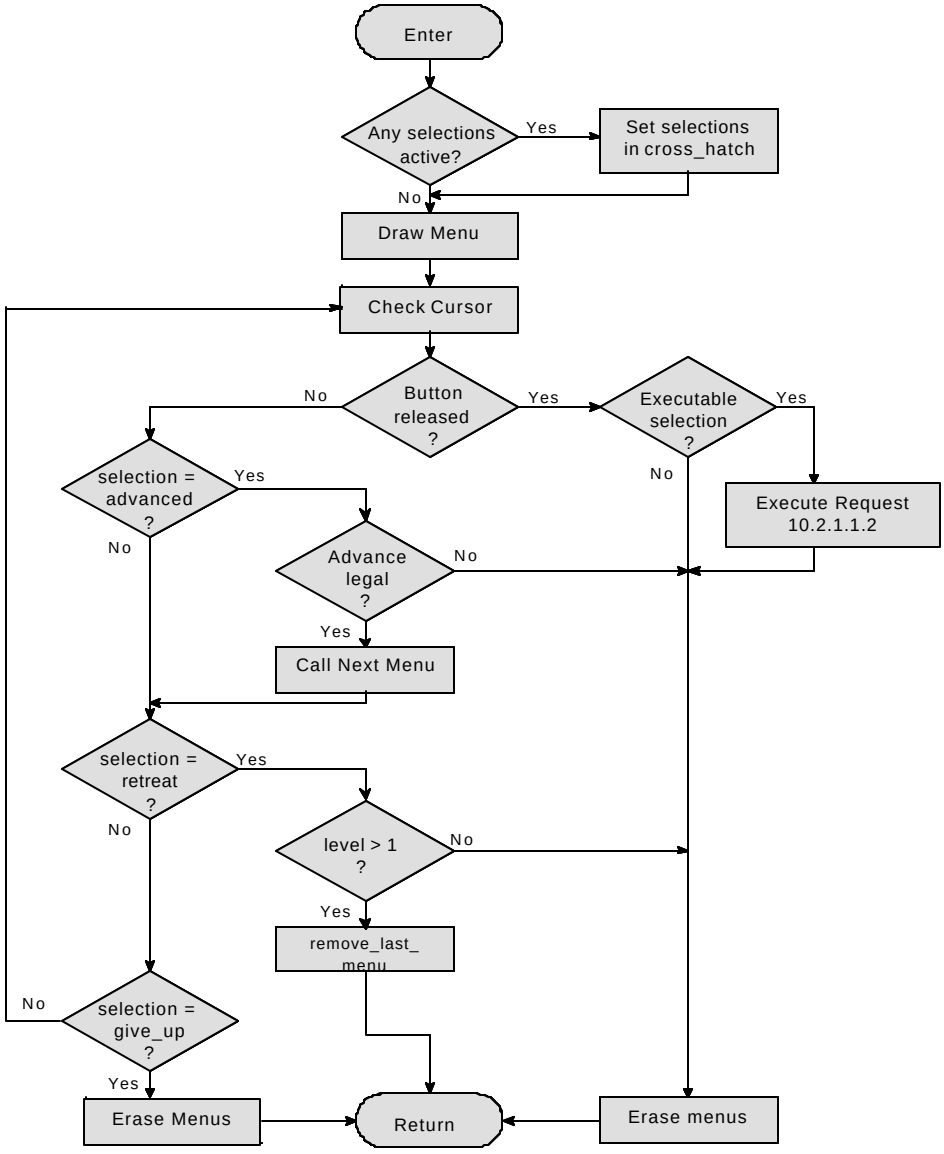

Figure 10-4. Logic for a Generic Menu Routine

10.2.1.1.1 The *Do Semicolon Command* Module

The *Do Semicolon Command* module is performed partly by the *command* routine and partly by the routines in *Execute Request*. The c*ommand* routine is invoked when the user hits the semicolon key. The *command* routine executes in the following sequence:

(1) prompts the user for the command entry by displaying a text-entry box on the display and reads the user's reply.

(2) extracts the first word from the reply and checks it against the list of legal commands; if not found, displays an error message temporarily and the exits the routine.

(3) If the command is legal, invokes the *Execute Request* routine for the associated command.

(4) If parameters are needed to execute the command, checks for them in the already entered text string, and if not present, prompts the user for them.

(5) When the parameters are known, performs the request, as described in the *Execute Request* module (Section 10.2.1.1.2).

The logic of the *Do Semicolon Command* module is shown in Figure 10-5.

10.2.1.1.2 The *Execute Request* Module

The *Execute Request* module consists of a large number of routines which performs the various functions that the user may invoke through keyboard commands, menu selections, and semicolon commands. The *Execute Request* routines are summarized in the following lists. The lists are organized according to the general category of routine.

The following routines perform functions related to displaying maps:

- *copy_bitmap* routine — makes it possible to print a copy of the display currently drawn on the screen.
- *display_lat_lon* routine — displays the latitude and longitude corresponding to the point on the display where the cursor is currently located. It is called in response to a keyboard period (**.**) command, either while drawing an experimental flight path or not, or by a middle mouse button while drawing an experimental flight path, or by the menu latitude/longitude command.
- *initialize* routine — initializes the display, ignoring the adaptation file, if any.
- *move_center* routine — changes the translation offset so that the display is centered over a different geographic point. It notes that the magnification has not changed so that other program code will not change the magnification. In

order to let the user type several **move** and/or **zoom** commands in quick succession and have the computer just redraw the screen once, the *test_keyboard type-ahead* feature has been added.

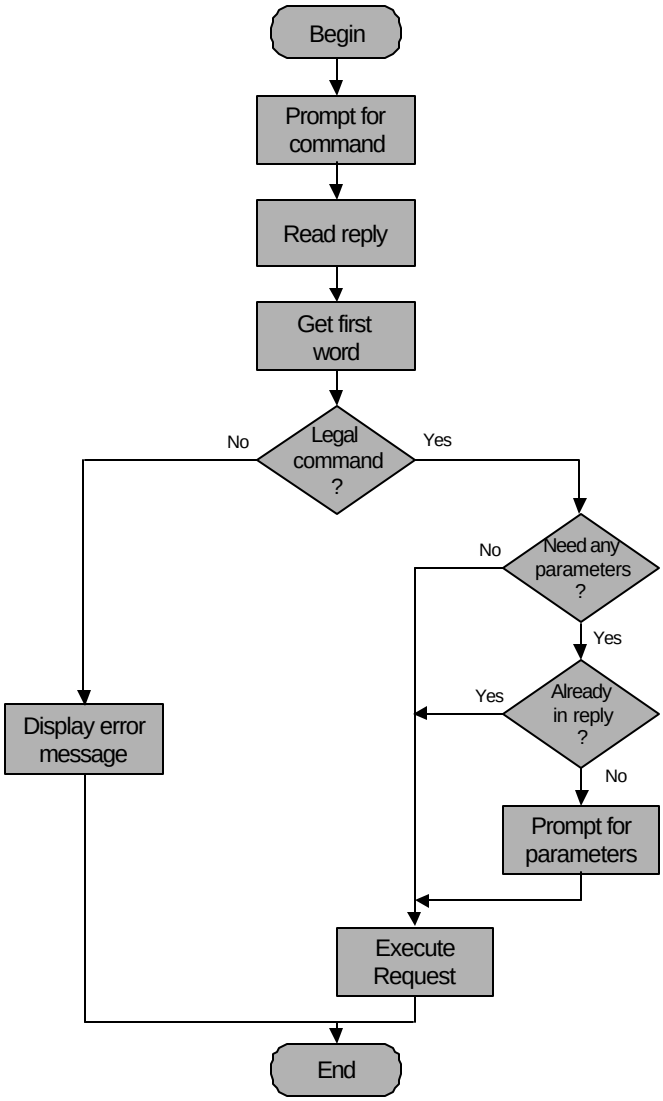

Figure 10-5. Logic for the Do Semicolon Command Module

- *pop* routine — changes the window so it takes up the whole screen. If it already takes up the whole screen, it returns the window to its original size.
- *specify_range_rings* routine — gets the specifications for the range rings.
- *super_move* routine — moves the center of the display to a specified place. The place name can be an airport, NAVAID, sector, or a place with a name as long as *moa Fort Bragg east*. This is the semicolon class version of the **move** command.
- *super_zoom* routine — moves the center of the display to a specified place with a specified magnification. This is the semicolon class version of the **zoom** command.
- *toggle* routine — turns an overlay on if it is off, or off if it is on. In either case, it puts the map overlay (e.g., a map of **high sectors**, or an overlay of **pacing airport** identifiers) into the set of overlays that are always on or always off, regardless of the magnification.
- *turn_off_range_rings* routine — turns off the range rings.
- *turn_on_range_rings* routine — turns on the range rings.
- *unzoom* routine — de-magnifies the display so that the screen shows a broader area.
- *zoom* routine — magnifies the display so that the screen shows an enlarged depiction of an area centered around the mouse cursor.

The following routines allow the user to manipulate display colors:

- *adjust_colors* routine — adjusts the six adjustable colors.
- *change_color* routine — called when a locator event has been sensed by the Apollo as a result of a cursor movement. If the cursor has been moved far enough to trigger a response, *change_color* moves the color box by erasing the present color box and drawing a new one in the new position.
- *choose_route_color* routine — allows the user to choose a new airplane highlighting color.
- *do_save_colors* routine — saves the color setup for each overlay on the disk in ASCII file format. It calls the *inquire_rgb_values* routine to convert the pixel color values into red, green, blue color intensities.
- *get_pixel_value* routine — converts the RGB triplets into a pixel color value.
- *get_colors* routine — reads the color file in ASCII format. It calls the *get_pixel_value* routine to convert the RGB triplets into a pixel color value.

- *initialize_colors* routine — sets the six adjustable colors to their standard values.

- *inquire_rgb_values* routine — converts the pixel color value into red, green, and blue color intensity values.
- *interpret_keystroke* routine — interprets mouse/trackball/touchpad buttons and/or keystrokes used to control the adjustment of the colors with the **adjust colors** command, as follows:
    - The left mouse button causes the colors to be set back to their standard values.
    - The right mouse button terminates color adjustment with the colors just as they are at the time the button is pressed.

  For the benefit of any Apollo nodes that might not have a mouse or trackball, the following keyboard characters can be used: **I** for initialize, **return** for accept the current values, or **Q** for quit the ASD program.
- *menu_background_color* routine — allows the user to choose a new menu background color.
- *menu_prompt_color* routine — allows the user to choose a new menu prompt color.
- *pick_color* routine — allows the user to pick a color.
- *read_binary_color_file* routine — reads the color file that was saved in binary record format.
- *restore_colors* routine — reads a color setup file off the disk and sets up the display the same way it was when the file was written.
- *save_colors* routine — stores the current color setup on a file on the disk, so the display can be set up the same way in the future.
- *select_background_color* routine — allows the user to specify the display background color. By default the color is pale green.
- *select_default_airplane_color* routine — allows the user to specify the default airplane color. The default airplane color is the color that an airplane would be if the user has not specified that it should be any other color. The user can specify other colors with the **selection color** command or the **default aircraft color** command. By default, the default airplane color is white.

The following routines allow the user to control what ETMS *site* he or she is getting data from:

- *show_site* routine — shows which site the *ASD* is currently connected to.
- *switch_site* routine — switches to another site. If it switches successfully, it displays a message telling the user that it has succeeded; otherwise, it displays

a message telling that it has failed.

- *switch_all_asds* routine — changes all *ASD* sites.

The following routines are used to control the display of flight data:

- *adjust_flagpole* routine — draws a slanted data block, if the user puts the cursor on a particular airplane icon, holds down the left mouse button, moves the cursor, and then releases the left mouse button. Slanted data blocks are permitted to allow the users to spread out the data blocks so they can read all of them, instead of having them drawn over one another.

- *adjust_future_flagpole* routine — draws a slanted data block, if the user puts the cursor on a particular proposed flight, holds down the left mouse button, moves the cursor, and then releases the left mouse button. Slanted data blocks allow the users to spread out the data blocks so they can read all of them, instead of having the blocks drawn over each other.

- *highlight_route* routine — finds the airplane icon nearest to the cursor and turns that airplane, its data block, and its flight path the specified color, so it can be distinguished from the other flight paths on the screen.

- *identify* routine — toggles the data block for the flight nearest to the cursor when the user hits the left mouse button.

- *m1_test* routine — responds to the user depressing the left mouse button according to the following conditions:

  (1) If the cursor is in the time bar interval, a time range selection message is displayed and the *ASD* waits for the second time interval to be selected.

  (2) If a report icon is selected, the selected report is displayed in a new text window.

  (3) If (1) and (2) do not apply, and if the button is released without moving, the airplane nearest the cursor has its data block displayed. If the data block is already displayed, the display is redrawn with that data block removed.

  If the cursor is moved before the left mouse button is released, the data block is drawn at the position where the cursor was when the mouse button was released.

- *nearest_airplane* routine — finds the airplane nearest to the cursor.

- *nearest_future_flight* routine — finds the proposed future flight nearest to the cursor.

- *normal_search* routine — searches for a particular flight by name without wildcards.

- *remove_flight_paths* routine — removes all the flights in the flight path linked list from that list and also from the list of highlighted flights.

- *search_and_rescue* routine — searches for a particular flight, turns on its data block, and puts the cursor on it, so a user can zoom in on it, if desired.

- *specify_lead_lines* routine — gets the specifications for the lead lines.
- *toggle_flight_paths* routine — turns flight paths on, if they are off, or off, if they are on.
- *toggle_lead_lines* routine — toggles lead lines on or off.
- *toggle_org_dest* routine — toggles origins and destinations on or off.
- *toggle_route_following* routine — toggles route following on or off.
- *toggle_route_of_flight* routine — toggles routes of flight on or off.
- *toggle_tail_lines* routine — toggles tail lines on or off.
- *toggle_trace* routine — toggles the **trace** feature on or off.
- *turn_off_data_block* routine — turns off the data block for a specified flight.
- *turn_off_flight_paths* routine — removes all flights from the flight path linked list and then redraws the flight icons with their flight paths turned off.
- *turn_off_highlight* routine — turns off color highlighting for a specified flight.
- *turn_on_flight_paths* routine — responds to user request to display flight paths. This routine causes the *ASD* to execute *request_flight_paths* to get the list of flights that need flight paths. The received list is sent to the *FTM*, which sends back the flight paths to the *ASD*, which then displays them.
- *un_highlight* routine — removes a flight from the list of color-highlighted flights.
- *un_flag* routine — removes a flight from the flagged flight linked list.
- *unzap* routine — turns an aircraft icon back on after it has been *zapped* with the \ command.
- *wild_card_search* routine — searches for a group of flights by name with wildcards.
- *zap* routine — removes an aircraft icon from the screen in response to the \ command.

The following routines allow the user to select subsets of flight data to be displayed:

- *select_color* routine — allows the user to display a specified subset of the flights in a specified color.
- *select_data_block* routine — displays data blocks for a specified subset of the flights in the air.

- *select_flag_parameters* routine — selects the parameters for displaying data blocks for a specified subset of the flights in the air.

- *select_flights* routine — sets up the specifications for displaying only a selected subset of the flights actually in the air. It also sets the time stamp fill color to red, (instead of yellow), so the user will know that not all the flights are being displayed.

- *select_parameters* routine — gets color selection parameters from the user.

- *select_visibility_parameters* routine — gets selection visibility parameters from the user.

The following routines are used to control the display of monitor/alert data:

- *display_report* routine — invoked after the user displays alerts with the **live alerts** command, selects a specific alerted element with the **examine** command, then requests a report with the **report** command. The system responds to these actions in the following sequence:

  (1) *ASD* asks the *TDB* for a list of flights to be expected in the selected element during the alert time period (by default, the next two hours).

  (2) *TDB* returns this list to the *ASD*, which passes the list to *ftp* to get the flight times and flight paths.

  (3) *ftp* returns flight times and paths to the *ASD*, which formats the data, creates a report pad, and displays the pad as a pane within the *ASD* window.

  The pane can be removed from the screen either by the standard Display Manager **exit** command or by using the *ASD* **report** command again (as a toggle).

- *do_airport_bar_chart* routine — calls *draw_airport_bar chart*, which actually draws the pacing airport bar chart, when the user has given an **airport** command, when a request has been sent to the *FTM*, and when the *FTM* has sent back the airport data.

- *examine_alerted_element* routine — responds to the **examine alerted element** command as follows:

  (1) If monitor alerts are not currently being displayed, it displays an error message and returns.

  (2) If alerts are being displayed, it tests whether there is a report being displayed in a pad pane, and if so, removes it from the screen.

  (3) turns off any time bar alarms that may still be on from examining some other alerted element.

(4) calls *draw_time_bar* to draw a new time bar for the newly selected element.

- *nearest_alert* routine — finds the alerted element (airport, fix, or sector) closest to the position of the cursor.

- *request_asp_data* routine — asks the *ASP* for data. It sets the **data_type** to **B** and then asks the *ASP* process for bar chart data. It is called by the **bar chart** or **report** command.

- *request_report* routine — reads the list of flights reported by the *ASP* as being predicted to be in the selected alerted element. Copies the list into a request block (8192 bytes long) to be sent to *ftp*, which will return the positions and flight paths for those flights.

- *select_alerts* routine — selects the types of alerts to be monitored.

- *set_alert_time* routine — prompts the user for the start and end of the time period for which alerts are authorized.

- *send_copy* routine — prompts the user for a node name, and sends a copy of the report to that node, if a report is currently being displayed when the user gives a **send copy** command. The report is re-formatted so that it can be printed out on 80—column width paper, if the user so chooses.

- *toggle_authorized_alerts* routine — toggles the authorization of the specified alert type on or off.

- *turn_green* routine — turns an element green in the alert list only. It does not re-draw the element in green. This does not happen until the screen is re-drawn, either because a new alert has been received, or because the screen has had to be re-drawn for some unrelated reason.

The following routines are used for the flight data replay feature. The replay functions also make use of the flight data routines:

- *date_time* routine — gets a date and time from the user.

- *get_dir_name* routine — prompts the user for the name of the directory from which to draw the replay data. Does some validity checking.

- *replay* routine — initiates replaying.

The following routines perform the script and adapt requests (adapt is just a special type of script):

- *adapt* routine — executes an adaptation file. It is called once when the *ASD* first starts up, and again any time the adaptation command is executed. The only difference between this routine and *script* is that this routine will always execute **/etms5/asd/adapt/adaptations/adapt_default** specifically, whereas *script* allows the user to specify what file should be executed.

- *adapt_command* routine — called by the semicolon class script command. It reads a specified adaptation file, not the standard

/etms5/asd/adapt/adaptations /adapt_default file, but **/etms5/asd/adapt/adaptations/filename**, where **file-name** is the name of the script file.

- *bomb_script* routine — closes the scripts file, displays the error message, and terminates the script processing by setting **end_script** variable to true.
- *escape* routine — allows a user to interrupt and terminate execution of a script file. (It is sometimes desirable to write a script, so that it goes into a loop and executes forever, until or unless someone stops it.)
- *follow_script* routine — executes a script or adaptation file.
- *GetIntegerValue* routine — reads and interprets an integer found in ASCII code in the script or adaptation file.
- *GetRealValue* routine — reads and interprets a real number found in ASCII code in the script or adaptation file.
- *interpret_color* routine — interprets the color name as specified in an adaptation, colors_default, or script file. Color names must be specified on the line cor-responding to where the prompt would come during interactive execution of the command, and must be spelled exactly as follows: **black**, **red**, **green**, **blue**, **cyan**, **yellow**, **magenta**, **white**, **pale green**, **dark green**, **reddish tan**, **brick red**, **pale blue**, **dull blue**, **khaki**, **brown**.
- *pre_move* routine — executes a semicolon **move** command from a script or adap—tation file.
- *pre_zoom* routine — executes a **zoom** command from a script or adaptation file.
- *script* routine — executes a script file. The difference between this routine and *adapt* is that this routine allows the user to specify what file should be executed, whereas *adapt* will always execute **/etms5/asd/adapt/adapt_default** specifically.
- *script_repeat* routine — used by script files to make an *endless* repetition of the script. It returns to the beginning of the script file and starts executing the script all over again. The user can stop the sequence by pressing the keyboard **esc** key.
- *script_wait* routine — used by script files. There are times when a script needs to wait a specified period of time (for example, to leave a display on the screen long enough for the viewer to get a good look at it). When a **script** is running, the **prompt** routine gets its reply by reading it from the script file, not by prompting the user through the screen and keyboard.
- *ScriptCommandError* routine — formats the diagnostic message and invokes the *bomb_script* routine to display the error message and end the script processing.

- *set_up_all_data_blocks_displayed* routine — executes a keyboard command (**show data blocks**) from a script or adaptation file.
- *set_up_lead_lines* routine — turns lead lines on or off from a script or adaptation file.

- *set_up_range_rings* routine — executes a semicolon **range rings** command from a script or adaptation file.
- *set_up_sua* routine — turns special use areas on or off from a script or adaptation file. An SUA can be specified, or all SUAs can be turned on or off together by the following switches:

  a — alert_areas

  m — moas

  p — prohibited_areas

  r — restricted_areas

  w — warning_areas

  + — all on

  - — all off
- *setup* routine — puts an overlay in the displayed set, or removes it if there was a **—** character in the script or adaptation file. In either case, the overlay is put in the override set, indicating that the overlay is not to be turned on and off automatically depending on the zoom level.
- *triad_test* routine — tests for three-letter keyboard commands *(triads*) in a script or adaptation file. The triads tested for are as follows:

  arr — for arrival fixes (to simulate the down-arrow-in-a-box key).

  dep — for departure fixes (to simulate the up-arrow-in-a-box key).

  pop — simulates the **pop** key.

The following routines perform functions related to drawing experimental routes and individual jet and Victor airways:

- *close_xroute* routine — resets the **drawing_experimental_route** to false and re-stores the cursor to the default blinking block.
- *delete_all_j_or_v* routine — deletes all the airways from the linked list in memory.
- *delete_route* routine — deletes one airway from the list of individual airways displayed on the screen.
- *display_route* routine — looks up an individual airway in the indexed database and displays it.
- *enter_airway_name* routine — adds an airway name to the list of airways displayed.

- *enter_xroute_node* routine — adds a node to the linked list of points on the experimental flight path being drawn.

- *remove_all_jv_airways* routine — goes through the list of individual airways currently being displayed on the screen and deletes them.

- *seek_navaid* routine — looks for the NAVAID nearest to the cursor and extends the experimental flight path to that point. It is called by a comma (**,**) or left mouse button while drawing an experimental flight path.

The following routines create, display, and maintain the lat/lon points:

- *dispose_latlong_list* routine — disposes the lat/lon linked list.

- *draw_latlong_list* routine — displays the lat/lon points in the linked list.

- *enqueue_latlong* routine — appends the lat/lon point to the end of the lat/lon linked list headed by the variable *latlong_list_head*. The end of the lat/lon list is pointed to by the variable *latlong_list_tail*.

The following routines enable the user to undo certain *ASD* commands such as MOVE, ZOOM, UNZOOM, and PROJECTION:

- *empty_undo_stack* routine — clears the undo stack.

- *pop_view* routine — restores the previous view settings from the undo stack.

- *stack_view* routine — saves the view center, projection type, and zoom_scale on the stack pointed to by the variable *undo_list_top*.

The following routines create, display, and reset the legend text lines:

- *draw_legend* routine — draws the existing legend text.

- *reset_legend* routine — clears the existing legend text.

- *set_legend* routine — stores the input text string into the variable *legend*, replaces any existing text, and redraws the new legend text.

The following routines allow the user to create and display weather maps:

- *button_test* routine — tests for the special weather-drawing mouse or trackball or touchpad button commands and executes them.

- *draw_symbol* routine — as the user moves the symbol about the display, stores the symbol at its current position after the user hits the left mouse button. Continues moving another copy of the symbol in response to the user's action. (See *remove_old_symbol* routine.)

- *draw_weather_map* routine — allows the user to draw a weather map on the screen by hand.

- *keystroke_test* routine — tests for the special weather—drawing keystroke

com-mands, and executes them.

- *remove_old_symbol* routine — As the user drags a symbol across the screen, erases the old symbol so a new one can be drawn in close proximity, thus giving the impression of motion.

10.2.2 The *Process Queued Messages* Module

The *Process Queued Messages* module is executed repeatedly as part of the main loop of the *Process Input* module of the *ASD*. On each pass through the loop, the *Process Queued Messages* module determines whether any new network message has arrived. If a new message exists, this module gets the data from the message and determines what to do based on the message contents. In most cases, the *Process Queued Messages* module invokes the *Draw Display* rou-tines to display the data to the user.

If the received message is a status reply of the previous user request, the *Draw Displays* module is called to display the status message. If the received message is a map update from the *FTM* process and *flights* is set in the *displayed* variable, the *Draw Displays* module is invoked to redraw the *ASD* window to show new traffic pattern for en-routed flights. If the received message is from the *ASP* process, the *Draw Displays* module is called to redraw the alerted elements, time bar, and bar chart (if displayed).

10.2.3 The *replay_test* Routine

The *replay_test* routine is invoked by the *Process Input* module via the *flight_check* routine on each pass through the main loop, if the global **replaying** flag is set. The *replay_test* routine checks whether the current time is past the **next_replay_time**.

If **true**, the *replay_test* routine searches the directory containing the replay data for the next **map** file in the replay time interval and calls the *draw_airplanes* routine to update the screen. If no more **map** files are found in the replay interval, a message is displayed, and the **replaying** flag is reset. If the **freeze** flag is set, no update is performed (see Figure 10-6).

10.2.4 The *test_icons* Routine

The *test_icons* routine is invoked during each loop of the *Process Input* module. The *test_icons* routine first checks whether any report window has been closed. If so, it invokes the *close_all _windows* routine to close all of the report windows and restores the *ASD* window to its original size. Then, the *test_icons* routine checks the cursor position against the positions of any dis-played icons and updates the global variables used to maintain the icons status. The actual displaying of the icons and handling of user operations on icons is performed in the *Respond To User Requests* module.

If any icons are displayed, the *test_icons* routine loops through the list of displayed icons. For each icon, *test_icons* determines if the cursor is currently in the icon box. If **true**, the **selected_icon** pointer variable is set to that icon record node, and the **icon_reversed** flag for

that icon is set **true**. If the cursor is not in the icon, the **icon_reversed** flag is set **false**. If the cursor is in no icon, the **icon_selected** pointer is set to **nil** (see Figure 10-7).

Error Conditions and Handling

Errors incurred during the *Process Input* module can be fatal or non-fatal. Non-fatal errors cause an error message to be displayed, but the *ASD* continues to execute. Fatal errors cause the *ASD* to terminate execution.

Before the *ASD* terminates its execution, it *cleans up* by invoking the *cleanup_handler* routine.

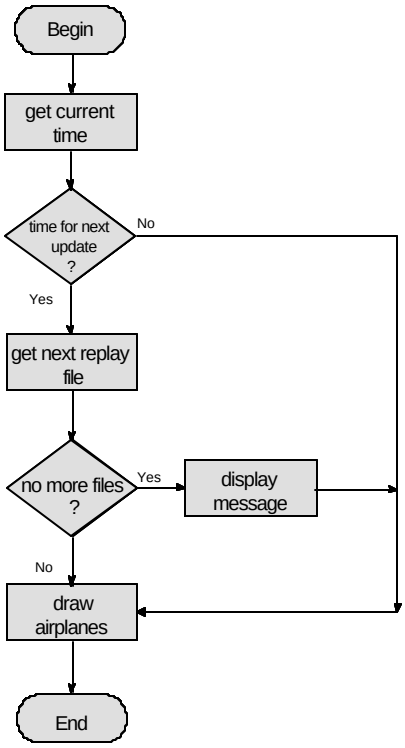

Figure 10-6. Logic for the replay_test Routine

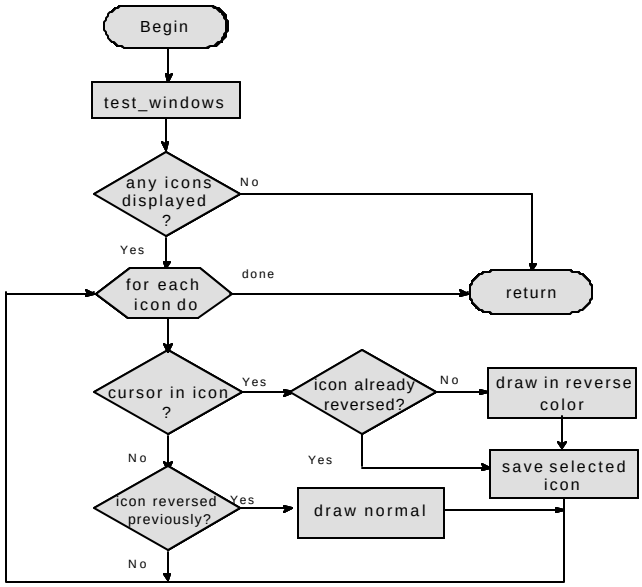

Figure 10-7. Logic for the test_icons Routine

10.3 The *Draw Displays* Module

The *Draw Displays* module contains routines that are invoked to generate the various types of displays available to the user. The *Draw Display* routines are invoked separately from many places within the *Process Input* module in response to a user request or data arriving from the network. The *Draw Display* routines are also invoked from the *Initialize* module.

Input

The *Draw Display* routines get the following data from the *Process Input* and *Initialize* modules:

- Display parameters — many variables including display type (color/bw), number of display color planes, display color set, current window size, maximum window size, font index array, color array, fill patterns, menu box sizes, cursor position, data displayed flags, map center, translation offsets, zoom scale, audible alarm settings, error log flag, and special mode flags (for experimental mode, large screen mode, etc.).
- Display data — dynamic data needed to draw the many displays that can be requested by the user. Includes *ASD* updates (active flight data), flight list and count reports, ARRD reports, weather reports, alerts, bar charts, alert reports, and alerted flight displays.

Output

The *Draw Display* routines do not generate output.

Processing

The *Draw Display* module is implemented as many separate routines. The routines are either invoked by other modules or by other *Draw Display* routines. The *Draw Display* routines are summarized in the following lists, organized by general type of function.

The following routine is used for the initial display:

- *display_title* routine — displays the program title, a disclaimer warning the user that this is an experimental prototype program, and the version number. The title is kept on the screen for ten seconds, unless the program is running in the privileged directory used only by the software developers, in which case it is displayed momentarily.

The following routines are used to draw the map displays:

- *arcsin* routine — created since Pascal has no *arcsine* routine; it uses *arctan*.
- *blt_view* routine — when a new flight data update becomes available, *blt_view*

draws a red light bar in the center of the top of the screen, creates a new display in a background bit-map, and copies it onto the screen. Thus, the airplanes appear to jump to their new positions.

The effect created by this routine is preferable to drawing the new flight data on the screen while the user is watching, It is not only aesthetically more pleasing, but it gives the user more time to study the old situation.

- *clear_parameters* routine — clears all the flight display parameters.
- *do_polyline* routine — reads a polyline from the **map.gpr** file (in its unique format) and draws it on the screen, scaled by *scale_x* and *scale_y*.
- *do_special_symbol* routine — draws the special *no data* and *no TZs* symbols on the display to indicate that one of the ARTCCs is not sending data. These symbols have two colors, (a black airplane or radar dish with the international prohibition symbol, a red circle and slanted bar, drawn over it).
- *do_super_move* routine — is called by *super_move* to do the actual move.
- *do_text_string* routine — reads a text string from the **map.gpr** file, (in its unique format), and draws it on the screen (scaled by *scale_x* and *scale_y*).
- *draw_background* routine — draws the background overlays of the display, but does not draw flights or monitor alerts. It is used by *view*, *blt_view*, *re_draw*, and *re_map*.
- *draw_latlong_list* routine — displays the lat/lon points in the linked list.
- *draw_legend* routine — draws the existing legend text.
- *draw_overlay* routine — a general purpose routine that draws any overlay. The overlays are: high_sectors, low_sectors, oceanic_sectors, superhigh_sectors, boundaries, artccs, jet_routes, victor_routes, arrival_fixes, departure_fixes, navaids, pacing_airports, terminals, alert_areas, moas, prohibited_areas, restricted_areas, warning_areas, individual_jet_and_victor_routes, experimental_route, weather_map, range_rings, and flights.

  While *draw_overlay* is a general purpose routine, there are a lot of special tests to provide for unique treatment for some of the overlays.
- *draw_range_rings* routine — draws the range rings.
- *DrawCircle* routine — draws the circle of a given radius at the specified center by computing the circle points and invoking the *DrawCirclePoints* routine.
- *DrawCirclePoints* routine — displays circle points of a given radius around the specified center point.
- *expand* routine — maintains the proportional integrity of a map display when the user executes the **zoom** command. Without this routine, for example, if a map of the U.S. were magnified by 2, only the northwestern one—quarter of the country would be displayed.

In order to zoom in and keep the magnified map centered upon the same center point, *expand* adds an expansion offset to **x** and **y**. This routine computes these offsets.

- *inverse_projection* routine — inverse of the *projection* routine (see next item). It converts x-y coordinates to latitude-longitude of the *Albers Equal Area Projec-tion*.

- *projection* routine — converts latitude-longitude values to x-y coordinates. The *Albers Equal Area Projection* is a conic projection that intersects the surface of the globe at two standard latitudes and is centered about one standard longitude (zlon).

- *re_draw* routine — re-draws the display, assuming that the translation and magnification have not been changed since last time.

- *re_map* routine — re-draws the display, assuming that the translation and magni—fication have not been changed since last time. It is a special routine for the dis—play manager to call whenever the screen needs to be re-drawn; for example, when the size or shape of the window has been changed.

- *refresh_asd_view* routine — checks if the *ASD*'s window is still being obscured. If so, it simply resets the *window_refresh_needed* to true and returns. Otherwise, it calls the *re_map* routine passing in the *window_moved* and *window_popped* values as parameters.

- *rr_scale_x* routine — a special scaling function for use with range rings. It is differ—ent from the scaling algorithm used in *view.pas*.

- *rr_scale_y* routine — a special scaling function for use with range rings. It is different from the scaling algorithm used in *view.pas*.

- *scale_x* routine — takes an **x** coordinate as output from the Albers projection algorithm and scales it according to the current window size, translation, and magnification.

- *scale_y* routine — takes a **y** coordinate as output from the Albers projection algorithm and scales it according to the current window size, translation, and magnification.

- *set_max_min* routine — calculates the maximum and minimum values of un-scaled **x** and **y** that can, (when scaled), fit in the current window.

- *set_text_displayed* routine — tests if text should be displayed at the current magnification. If certain classes of text are displayed at too small a magnification, the screen will be unreasonably cluttered.

- *set_view_center* routine — restores the view center to the previous one before

the view is re-drawn to keep the view center from changing unexpectedly.

- *setup_color_node* routine — sets up the initial conditions for a color node.
- *skip_text_string* routine — skips over a text string from the **map.gpr** file, (in its unique format), and does not draw it on the screen.
- *text_in_box* routine — draws a number in a box to label the range rings.
- *translate* routine — in addition to the expansion offset, adds amounts to **x** and **y** to slide the picture over to the arbitrary center point that the user wants.
- *view* routine — draws the display, assuming that the magnification or translation may have been changed. It lets the user watch the flights being drawn on the screen.
- *unscale_x* routine — inverse of *scale_x*. It takes scaled coordinates and calculates the original coordinate that it must have been when it came out of the Albers projection.
- *unscale_y* routine — inverse of *scale_y*. It takes scaled coordinates and calculates the original coordinate that it must have been when it came out of the Albers projection.
- *window_refresh_handler* routine — saves the window redraw conditions into the global variable *window_popped* and *window_moved*, and sets the *window_re-fresh_needed* variable to **true**.

  This routine should be specified in the gpr_$set_refresh_entry call so that the system will invoke the *window_refresh_handler* whenever the *ASD* requires up-date as a result of window resize, move, or pop.

The following routines are used to support the adjust colors command:

- *draw_box* routine — draws a colored box to represent the intensity value of the specified color.
- *draw_boxes* routine — draws a colored box to represent the intensity value of each of the six adjustable colors available on the least sophisticated Apollo computers likely to run the *ASD* program.
- *erase_box* routine — erases a color box by copying the background onto the dis-play in the place where the color box had been. The color box appears to move smoothly across the screen.
- *erase_boxes* routine — erases all the color boxes.

The following routines are used to draw the flight data displays:

- *actype_qualifies* routine — tests whether an aircraft type falls within a particular set of selection criteria.

- *adjust_flagpole* routine — draws a slanted data block. Slanted data blocks allow the users to spread out the data blocks so they can read all of them, instead of having them drawn over each other.
- *adjust_for_id_overlap* routine — builds a linked list of all the flights for a particular departure point and allocates a different x-y position at which to display each flight's identifier. Usually, several flights are waiting to take off from any given airport. This routine avoids overprinting their data blocks at a single point on the screen.
- *airplane_selected* routine — tests if a flight falls within a particular set of selection criteria.
- *airplane_visible* routine — tests if a flight falls within a particular set of visibility criteria.
- *altitude_qualifies* routine — tests if a plane's altitude falls within a particular set of selection criteria.
- *check_flight_paths* routine — checks all the flights in the flight path linked list, first to see whether they are already in the air. If so, they are displayed as flying aircraft; otherwise, they are displayed as waiting on the ground.
- *compute_altitude* routine — interprets the plane's altitude. In some cases, a flight may have a block of altitudes assigned to it, in which case *compute_altitude* uses logic to see if any part of its block of assigned altitudes falls within the range the program is testing for.
- *data_block_displayed* routine — tests if there is any data block displayed.
- *determine_color* routine — tests if a flight should be displayed in a special color.
- *display_aircraft_type* routine — displays the aircraft type in the data block.
- *display_buffer_size* routine — displays an error message when a record is found to be the wrong length. Since the route file has a fairly complicated format, it is necessary to do a lot of checking to make sure it is being read correctly.
- *display_fld10* routine — displays the Field 10 of the plane's flight plan, if it is known; otherwise, it displays **unkw**.
- *display_org_dest* routine — looks up a plane's origin and destination and displays them in its data block.
- *draw_airplane* routine — draws an airplane icon.
- *draw_airplanes* routine — draws the airplanes as a background bitmap, then

copies them onto the screen so that they all appear at once.

- *draw_data_block* routine — draws the data block for a flight.
- *draw_lead_line* routine — draws a lead line for a flight. A lead line points out in front of the airplane icon a specific distance, based either on how far the plane will move in a specified time, or based on a specified distance.
- *draw_time_stamp* routine — draws a time stamp on the upper left-hand corner of the screen.
- *fill_fld10* routine — draws the Field 10 of the flight plan for the specified flight. If this information is not available, it sends the message **unkw**.
- *fix_qualifies* routine — tests whether a fix falls within a particular set of selection criteria.
- *follow_route* routine — plots a flight path, given a list and count of waypoints.
- *keyword_qualifies* routine — tests if a keyword falls within a particular set of selection criteria.
- *on_the_ground* routine — draws a data block for a flight that is still on the ground, waiting to take off. The regular data block for a flight in the air has places for information that is not relevant to a flight that is still on the ground, for example, the altitude and ground speed; hence, this simplified data block was developed.
- *light_bar* routine — draws a small red bar at the center of the top of the screen to indicate that flight data is being updated. This informs the user that the computer may be delayed in responding to commands.
- *minutes_to_destination* routine — looks up the plane's destination and calls *minutes _to_point* to calculate how long it would take to get there. If it cannot find the destination in **map.gpr**, *minutes_to_destination* returns zero.
- *minutes_to_point* routine — calculates the number of minutes it would take for a flight to reach a given point, if the plane were to continue flying at its present speed until it reached that point. This routine makes no allowance for flight performance profile, altitude, slowing down to land, etc.
- *org_dest* routine — displays the origin and destination of the specified flight.
- *origin_qualifies* routine — tests if a plane's origin falls within a particular set of selection criteria.
- *prefix_qualifies* routine — tests if a prefix falls within a particular set of selection criteria.

- *read_flight_data* routine — reads the **map** file and looks at all the airplanes. It then displays all the flights that are supposed to be displayed according to the visibility and selection criteria at the time.
- *re_draw_data_blocks* routine — re-draws all the flight data blocks without drawing all the rest of the display.
- *sector_qualifies* routine — tests if a sector falls within a particular set of selection criteria.
- *suffix_qualifies* routine — tests if a suffix falls within a particular set of selection criteria.
- *time_stamp* routine — draws a time stamp on the upper left-hand corner of the screen if live data are being displayed; draws a date block in the upper right-hand corner, if the *ASD* is doing a replay.

The following routines are used to generate the monitor/alert displays:

- *add_minutes* routine — adds integer minutes to a time value in **time_$clock_t** format.
- *adjust_for_overlap* routine — in the event that two alerted elements are in the same place on the display, or at least near enough so that their symbols overlap, moves the newer one downward by 15 pixels.
- *display_line* routine — displays a line of text on the screen.
- *do_asp_bar_chart* routine — draws a bar chart, using data already furnished by the *ASP*.
- *draw_airport_bar_chart* routine — draws a pacing airport bar chart and keeps it displayed until the user depresses any key on the keyboard.
- *draw_airport_bars* routine — draws the bars for an airport bar chart.
- *draw_airport_or_fix_alert* routine — draws an airport or fix alert on the map. The only difference between an airport alert and a fix alert is the symbol displayed. Sector alerts are drawn in a much more complicated manner and require a dif-ferent routine.
- *draw_arrow* routine — draws a red arrow on the time bar to point to the present Universal Coordinated Time.
- *draw_bar* routine — draws a single bar for a bar chart.
- *draw_bar_chart* routine — invoked for drawing a bar chart. It calls the appropriate lower level routine according to the type of bar chart needed.
- *draw_bar_chart_airport* routine — draws a bar chart for an airport.

- *draw_bar_chart_fix* routine — draws a bar chart for a fix.
- *draw_bar_chart_sector* routine — draws a bar chart for a sector.
- *draw_color_block* routine — fills the time bar strip of color blocks, one for each time interval in the time span. If no element has been examined , all blocks appear in the background color. If an element has been examined, the alerted intervals appear in red, and the other intervals appear in green.
- *draw_colored_sector* routine — draws an alerted sector in the appropriate color, depending on which kind of alert is posted for that sector. The sector boundaries have already been stored in the alert node; a pointer has been passed to this routine.
- *draw_pacing_airport_bars* routine — draws the bars for a pacing airport bar chart.
- *draw_regular_bars* routine — draws the bars for a regular (i.e. not an airport) bar chart.
- *draw_time_bar* routine — draws the time bar for the Monitor/Alert feature.
- *legend* routine — displays the legend that explains the meaning of the bars on the bar chart.
- *remove_bar_graph* routine — removes a bar chart from the screen. It does it by simply re-drawing the display with the drawing-bar-chart switch off.
- *set_alert_colors* routine — sets up the text font, text value, draw value, and fill color according to the type of alert.
- *set_up_stripes* routine — sets up the fill pattern and color for cross-hatching in alerted sectors displays. Alerted sectors are drawn with stripes shown in different cross-hatch patterns, depending on whether the sector is a high sector, low sector, or superhigh sector, and different colors, depending on the alert level of the sector.
- *sound_all_alarms* routine — scans the entire list of alerted elements, then displays classes of elements that the user has authorized to be displayed. The **select alerts** command is used to specify the types of alerted elements that may be displayed.
- *time_bar_tick_mark* routine — draws a tick mark for the time bar.
- *time_ok* routine — checks all the time intervals from **blue_line_start_interval** to **blue_line_end_interval** to make sure that at least one of the time periods has a red alert.
- *time_to_x* routine — converts a time value to an **x-**coordinate value.

- *triangle* routine — draws a triangle to represent a fix alert.

The following routines are used to draw experimental routes and selected jet and Victor airways.

- *follow_chain* routine — follows the linked list of individual airways to be displayed on the screen; displays each airway in turn.
- *follow_xroute* routine — follows the linked list of points on the experimental route and draws it on the screen.
- *jv1* routine — draws an individual airway.
- *label_airway* routine — puts a label on the middle of a segment of an airway.

The following routines support the menu functions:

- *contrasting_x* routine — draws an **x** across a menu color box in a color contrasting to the currently selected color.
- *draw_color_box* routine — invoked when the user opts to use the keyboard color boxes to specify a color, rather than using the menu color palette. This routine draws one color box. At present, the user can select from among 16 colors.
- *draw_menu_box* routine — draws one menu box.
- *pop_color_block* routine — pops the 16 keyboard color boxes up on the screen as a block.
- *pop_menu* routine — pops the menu up on the screen and makes the cursor active.
- *position_menu* routine — determines the position at menu levels are drawn, according to the following conditions:
  - If the cursor is away from the edges of the ASD window, positions menus near the cursor.
  - When the cursor is near one of the four edges of the window, positions menus to fit entirely inside the window, which requires menus to overlap a previous menu or menus.
- *remove_all_menus* routine — removes all the menus.
- *remove_last_menu* routine — removes the last menu that was put up on the display. This happens whenever the cursor is moved up, down, or backwards outside of the area covered by the menu.
- *reverse_values* routine — turns the menu box that has the cursor in it *red* instead of its standard color to highlight the option being selected.

- *shadow* routine — draws a *shadow* below and to the right of the menu. It makes the menu appear to be floating in the air above the surface of the screen, thus making it easier to see.

The following routines support the **show** command processing:

- *delete_sho_location* — removes individual identifiers displayed by the show command.
- *draw_individual_element* — checks whether a selected identifier is currently being displayed by the show command. If not, the identifier is drawn.
- *draw_selected_map_elements* — draws the list of selected identifiers.

- *enter_sho_location* — creates the list of identifiers specified by the user.
- *get_word2* — searches for identifiers within a list for a — character. This identifies items the user wants to delete.
- *init_show_table* — initializes the list of different types of overlays that can be displayed by the show command.
- *lookup_place* — finds the identifiers that were entered by the user among the list of actual identifers and checks if they are valid.
- *remove_all_places_shown* — removes all the identifiers displayed by the show command.
- *show_places* — main loop that checks if the user is entering identifiers or deleting identifiers.

The following routines support the semicolon command processing:

- *deallocate_prompt_background_bitmap* routine — deallocates the prompt back-ground bitmap, which had been used to store the part of the display that was overwritten by the message.
- *display_and_scroll* routine — when the user enters input in response to a prompt that is too small to contain the input, this routine scrolls the input to the left. This action can result in the input appearing outside the window altogether.
- *display_error_message* routine — displays an error message and waits until the user types some character from the keyboard. This routine is used to ensure that the user notices the error message.
- *display_message* routine — displays a message on the screen. Before displaying the message the routine copies the present contents of the part of the screen where the message will be displayed to a background bitmap, so it can be copied back later.
- *display_msg_2_seconds* routine — displays a message for two seconds; then, restores the screen to its former display.
- invalid_response routine — displays a message: Invalid response. Hit space bar.
- *prompt* routine — prompts the user for information and accepts the reply.
- *remove_message* routine — removes a message from the screen by restoring the previous contents of that part of the screen from the prompt background bitmap.

The following routines are used to draw the weather products from the weather server:

- *draw_erl_polyline* routine — draws either a filled polygon or polylines. If the polygon represents a hole, fills it with the background color and draws the contour in level 1; otherwise, draws the polylines.

- *draw_erl_multiline* routine — draws a set of disjoint line segments.
- *draw_erl_text* routine — draws the weather text.
- *erl* — maps the weather file into memory, checks for any file error, parses, and displays the data.
- *lightning* routine — draws the lightning weather symbol.
- *select_font* routine — sets the text font to the font number read from the file.
- *set_absolute_addressing* routine — determines the whether to use absolute addressing. When frame addressing is in used, all coordinates refer to a location in the frame and that items will stay at the same location on the visible frame regardless of the amount or location of the graphics being displayed upon.
- *set_character_magnification* routine — sets the character magnification read from the file.
- *set_character_spacing* routine — sets the character spacing. Character spacings are defined as a percentage of the default for that font and magnification.
- *set_character_style* routine — sets the character style to the one read from the file.
- *set_display_class* routine — sets the display class to the one read from the file.
- *set_draw_color* routine — gets the color intensity value.
- *set_text_centering* routine — controls what part of a text string is actually placed at its location coordinates. Left to right is always considered to be in the update dir—ection. Top left (00) is the default.
- *set_text_direction* routine — sets the text direction to the one read from the file.
- *set_relative_addressing* routine — sets to the relative addressing defined in the file. This is a mode where coordinates that follow refer to an offset from the origin.
- *set_vector_texture* routine — sets the line texture. The texture map is a bit map that defines the line texture. Each bit corresponds to one pixel along a line. The pattern length is the number of bits to use in the texture map (starting with the least significant bit) before repeating it, and can range from 0 to 16 decimal.
- *specify_input_size* routine — sets the frame size. It is used to compute the aspect ratio. If used, the first directive in the weather product must specify the frame size.

- *wx_graphics* routine — invokes the *erl* routine to read and display the weather product data in the **wx_maps** directory.

- *reroute_user_line_style* — sets the type of line to be drawn (i.e., dotted or solid).
- *reroute_solid_line* — draws a solid line which is defined within the reroute file.
- *reroute_line_thickness* — sets the thickness of the line to be drawn with a reroute file.
- *reroute_text_line_color* — sets the text and line color for a reroute weather file.
- *reroute_text_font* — sets the font for any text within the reroute weather file.

The following routines are used to draw the weather maps:

- *delete_old_string_chain* routine — deletes the old string chain, because a new one is being created.
- *delete_old_weather_chain* routine — deletes the old weather symbol chain, because a new one is being created.
- *delete_text* routine — deletes a string of weather text from the screen; replaces it with the background that used to be there before the text was typed in.
- *draw_circle* routine — draws a circle; it is used for drawing warm, stationary, and occluded fronts. On the displayed front, it looks like a semi-circle, but that is only because half of the circle is hidden behind the rectangular segment.
- *draw_fronts* routine — draws lines, areas, and all four kinds of fronts.
- *draw_menu* routine — draws the menu of weather symbols from which the user can make selections.
- *draw_segment* routine — draws one tiny segment of a front.
- *draw_symbol* routine — draws a specified symbol at a specified x-y location on the screen.
- *draw_weather_symbol* routine — scales the position coordinates and displays the hand-drawn weather symbol by calling the *draw_symbol* routine.
- *draw_text* routine — draws typed text on the screen as the user types it in; enters it into the string chain at the same time.
- *draw_tick_mark* routine — draws a tick mark, that can be either a triangle or a circle.
- *draw_triangle* routine — draws a triangle; it is used for drawing cold, stationary, and occluded fronts.
- *enter_new_front* routine — adds a new front to the end of the front list.

- *enter_string* routine — enters a text string into the text string chain.

- *enter_symbol* routine — enters a weather symbol into the chain.
- *erase* routine — erases the weather symbol or text nearest the cursor position.
- *erase_front* routine — erases a front, then restores the background that was pre-viously behind it.
- *find_nearest_text* routine — finds the string of weather text nearest the cursor.
- *follow_string_chain* routine — follows the weather text strings that are stored in a linked list, and draws the weather map.
- *follow_weather_chain* routine — follows the weather symbols that are stored in a linked list, and draws the weather map.
- *front* routine — draws fronts, areas, and lines.
- *plus_90* routine — called only by *draw_segment*. It draws a rectangle, centered around the segment. The name is derived from the fact that the two short sides of the rectangle are at a 90-degree angle to the long sides.
- *put_front* routine — displays a weather front.
- *read_weather* routine — reads in a weather data file from the disk and draws the weather data symbols on the screen.
- *restore_symbols* routine — goes through the chain of weather symbols and draws them on the screen.
- *save_background* routine — saves the entire display bitmap onto the background bitmap.
- *set_appropriate_font* routine — sets up the appropriate font for the weather sym—bols. There are two fonts; one very small set for very small windows and a normal set for normal windows.
- *set_symbol_color* routine — sets the color for a weather symbol. If it is **Q** (the big capital L for a *low*), it appears in red. If it is **R** (the big capital H for a *high*), it appears in blue. Otherwise, the weather symbols appear in the same color a flight icon would.
- *write_text* routine — writes text on the screen as the user types it in from the keyboard; stores it in memory in the string list at the same time.
- *write_weather* routine — writes all the weather data out onto a file on the disk, so the user can read it back in later, if desired.

Error Conditions and Handling

Errors incurred during the *Draw Displays* module can be fatal or non-fatal. Non-fatal errors cause an error message to be displayed, but the *ASD* continues to execute. Fatal errors cause the *ASD* to terminate execution.

Before the *ASD* terminates its execution, it *cleans up* by invoking the *cleanup_handler* routine. See Section 12.1 for details of the clean-up processing.

10.4 *ASD* Source Code Organization

This section describes the source code used in building the executable version of the *ASD*. The source code resides in Pascal/C files. Each file contains one or more functional units called a *routine*. A routine is implemented as either a Pascal function or procedure. The Pascal/C files have been organized as elements in a Domain System Engineering Environment (DSEE) library called **map_lib**. Most modules are written in Pascal but some are written in C. Hence, both the Pascal compiler and the C compiler are required to compile the source files.

Before the *Aircraft Situation Display* can be executed, the following DSEE's commands must be issued in order to compile the appropriate files:

(1) **set system map_sys**

(2) **set model map.sml**

(3) **set library map_lib**

(4) **edit thread -mod**

Type in the following lines and press the **exit** function key to save the model thread and exit the Domain's editor:

—reserved
[asd_etms_5.1.0] —when_exists

(5) **set model map.sml**

NOTE: The DSEE's command *set model* must be executed after every change in the build model thread via the *edit thread -mod* command.

(6) **edit thread**

Type in (or uncomment) the appropriate lines and press the **exit** function key to save the build thread and exit the Domain's editor. The following shows configuration for a *prototype*:

```
## PROTOTYPE VERSION — for FAA evaluation release
—FOR asd_version.pas —USE_OPTIONS —subchk —comchk —opt 0 —config prototype —config whatstring
—FOR map.pas         —USE_OPTIONS —subchk —comchk —opt 0 —config prototype
—FOR asd_net_add.pas —USE_OPTIONS —subchk —comchk —opt 0 —config prototype
—FOR prompt.pas      —USE_OPTIONS —subchk —comchk —opt 0 —config prototype
—FOR ?*.pas          —USE_OPTIONS —subchk —comchk —opt 0
—FOR ?*.c            —USE_OPTIONS —subchk —comchk —opt 0
## BETA VERSION— for BETA test sites (key field sites)
#—FOR asd_version.pas —USE_OPTIONS —subchk —comchk —opt 0 —config beta —config whatstring
#—FOR map.pas         —USE_OPTIONS —subchk —comchk —opt 0 —config beta
#—FOR asd_net_add.pas —USE_OPTIONS —subchk —comchk —opt 0 —config beta
#—FOR prompt.pas      —USE_OPTIONS —subchk —comchk —opt 0 —config beta
```

```
#—FOR ?*.pas          —USE_OPTIONS —subchk —comchk —opt 0
#—FOR ?*.c            —USE_OPTIONS —subchk —comchk —opt 0
## RELEASE VERSION — for FIELD RELEASE — operational version
#—FOR asd_version.pas —USE_OPTIONS —subchk —comchk —opt 0 —config whatstring
#—FOR map.pas         —USE_OPTIONS —subchk —comchk —opt 0
#—FOR asd_net_add.pas —USE_OPTIONS —subchk —comchk —opt 0
#—FOR prompt.pas      —USE_OPTIONS —subchk —comchk —opt 0
#—FOR ?*.pas          —USE_OPTIONS —subchk —comchk —opt 0
#—FOR ?*.c            —USE_OPTIONS —subchk —comchk —opt 0
## DEBUG VERSION
#—FOR map.pas         —USE_OPTIONS —subchk —comchk —opt 0 —config prototype —config whatstring —dba
#—FOR asd_net_add.pas —USE_OPTIONS —subchk —comchk —opt 0 —config prototype —dba
#—FOR prompt.pas      —USE_OPTIONS —subchk —comchk —opt 0 —config prototype —dba
#—FOR asd_version.pas —USE_OPTIONS —subchk —comchk —opt 0 —config prototype —dba
#—FOR ?*.pas          —USE_OPTIONS —subchk —comchk —opt 0 —dba
#—FOR ?*.c            —USE_OPTIONS —subchk —comchk —opt 0
## Version 5.1
—reserved
[asd_etms_v5.1.0]     —when_exists
```

(7) **build map.exec**

To create a release after building the *ASD* with new changes, execute the following DSEE's command:

(8) **create release release_directory_name —from map.exec!*timestamp* —exp ?***

10.5 *ASD* Version Naming Conventions

The following describes the conventions for naming the *ASD* versions.

10.5.1 Naming the Build Version in DSEE

- For FAA evaluation release, specify the process name (e.g. *asd*) followed by the prototype version number (e.g. *v5.0.p68)*. For instance, after executing the DSEE's build command, invoke the following DSEE's command to name the build version to *asd.v5.0.p68:*

  name version map.exec!*timestamp* asd.v5.0.p68

- For field release, specify the process name (e.g. *asd*), the *_etms_* tag, and the re-lease version number (e.g. *v5.1.0)*. For instance, after executing the DSEE's build command, invoke the following DSEE's command to name the build version to *asd_etms_v5.1.0:*

  name version map.exec!*timestamp* asd_etms_v5.1.0

10.5.2 Compiler Options

The appropriate compiler options that are used to name the *ASD* version are shown in Table 10-1.

Table 10-1. Compiler Options

Version Naming via Compiler Options	
Compiler Options	**Explanations**
—config whatstring	Includes *whatstring* variable in various modules. *ASD* version is defined in the *asd_version.ins.pas* which is included in four modules. In order to prevent **what** command from returning version number four times (once for each module it is included within), use the *–config whatstring* option on only one module: *asd_version.pas.*
—config test	Builds the test version of the *ASD*. The startup title page identifies the software as PROTOTYPE SOFTWARE FOR IN –HOUSE TEST and displays test version number (baseline version with the .test appended (e.g. 5.0.p68.test).
—config prototype	Builds the prototype version of *ASD*. The startup title page identifies the software as PROTOTYPE SOFTWARE OR EVAULATUION and displays the prototype version number (e.g. 5.0.p68 would be the 68th prototype version built).
—config beta	Builds the beta version of the *ASD*. The startup title page identifies the software as PROTOTYPE SOFTWARE FOR EVALUATION and displays the beta version number (e.g. 5.0b3 would be the third version delivered to beta sites).
(none)	Builds the field release version of the *ASD*. The startup title pag does not identify the software as prototype. Only the field release version number is displayed (e.g. 5.1.0).

NOTE: There are other compiler options for *adr* and *worldwide*. They are defined in the *asd_version.ins.pas* include file.

10.6 *ASD* Data Structure Tables

The Pascal record definitions of all data files are located in **map.ins.pas**.

10.6.1 The */etms5/asd/data/map.gpr5* File

The **/etms5/asd/data/map.gpr5** is a homegrown graphics metafile. The file begins with a directory table that points to the starting position of each *overlay*, i.e., each of the independent sets of items that can be drawn on the display, such as state and national boundaries, ARTCC

boundaries, sector boundaries, etc. The data that the program finds at the address pointed to by the directory consists of *keycodes* followed by data. There are three keycodes:

- **polyline,** consisting of
    - a polyline header, containing the size; and the minimum and maximum values of x and y that occur in the polyline. These values in the header make it possible to tell immediately whether any part of the polyline lies within the part of the display space that is currently displayed on the screen. If none of it is to be displayed, the data can be passed over with no further processing.
    - the data points, in unscaled Albers Equal Area Projection coordinates.
- **text_string**, consisting of:
    - **text_header**, containing the size of the data stream in bytes; and the x and y coordinates, in unscaled Albers Equal Area Projection coordinates, where the text is to be displayed.
    - the text string.
- **end_of_overlay**, which consists of just the keycode itself.

10.6.2 Flight Data

Flight data comes from the *FTM* process in the form of **map** file. The **map** file record format is
shown in Table 102.

Table 10-2. Map File Record

id			id_record
altitude			word8
destination			dest_name_t
aircraft_type			actype_t
flight_index			integer32
eta			integer32
seek_key	All 1's if not rte record written.		integer32
alt1			integer
alt2			integer
x			integer
y			integer
old_x			integer
old_y			integer
lat			integer
lon			integer
old_lat			integer
old_lon			integer
heading			integer
groundspeed			integer
cta			integer
flags			0..15
source_flags			integer
remarks_flags			integer
geo_filter			integer
filed_alt			integer
filed_alt2			integer
filed_speed			integer
filler			array 1..13 of integer
center_id			char
altitude_type			char

lat_lon_heading			char
symbol	For normal aircraft: a = headed north b= headed northeast c = headed east d = headed southeast e = headed south f = headed southwest g = headed west h = headed northwest For hollow aircraft: i = headed north j = headed northeast k = headed east l = headed southeast m = headed south n = headed southwest o = headed west p = headed northwest Display patterns: . = dot ^ = small circle — = large circle		char
waypoints	# of 4 bytes waypoints		char
sectors	# of 6 bytes sectors		char
fixes	# of 6 bytes fixes		char
airways	# of 6 bytes airways		char
centers	# of 3 bytes center identifiers		char
route_bytes	# of bytes		char
acenter	Arrival code		char
dcenter	Departure code		char

last_update	T = last update was TZ D = last update was DZ F = last update was FZ U = last update was UZ A = last update was AF S = last update was FS L = last update was AZ R = last update was RS Z = last update was RZ O = last update was TO W = last update was TA Y = last update was FY C = last update was RY E = last update was EDCT	T,D,F,U,A,S,L,R,Z, O,W,Y,C,E	char
air_cat			char
prefix_digit			char
prefix_char			char
suffix_char			char
ghost_to_rte			boolean

10.6.3 Alert Data

Alert data comes from the *ASP* and the *TDB* are in the form of **Global Alert**, **time bar data**, and **bar chart data** files. The names of these files are derived from the network messages that are sent from the *ASP* and the *TDB* to the *ASD*.

10.6.3.1 The Global Alert File

The the alerted elements in the **Global Alert** file are described in Table 10-3 .

Table 10-3. Global Alert File

Global Alert File			
Library Name: map_lib		**Element Name:** map.ins.pas	
Purpose: To pass global alerts from the ASP to the ASD.			
Data Item	**Definition**	**Unit/Format**	**Var.Type**
color	Color of alert (red, green, or yellow).	R,G, or Y	char
e_types	element type		set of alert_type
name	name of element	10 alphanumerics	char 10_t
x			integer
y			integer
source			alert_data_type
alarms	alarms (one per period)		alarm_node_ptr
non_current_alarms			alarm_node_ptr
audible_alarm_flag			boolean
next_node			alert_node_ptr

10.6.3.2 The Time Bar Data File

The **time bar data** file gives a list of elements, time intervals, and flights. Each element record is followed by a fixed number of time interval records. The format of the **time bar data** file record is shown in Table 10-4.

Table 10-4. Time Bar Data File Record

time_bar_record			
Library Name: map_lib		**Element Name:** map.ins.pas	
Purpose: To list elements, time intervals, and flights.			
Data Item	**Definition**	**Unit/Format**	**Var.Type**
time_bar_window			gpr_$window_t
time_bar_interval_count	# of 15 minute intervals		integer
asp_time_bar_interval_count	# of 15 minute intervals		integer
time_bar_interval_length			integer
time_bar_interval_space			integer
time_bar_limits			time_span

10.6.3.3 The Bar Chart Data File

The bar chart data file contains detailed list of arrivals, departures, and capacities for each *nas_event_t* case. In the case of the airports, the record will have the data for arrival/departure capacities, number of active arrivals/departures, and total arrivals/departures. In the case of the fix crossings, the record will have the data for the capacities of various fixes (i.e. low, high, and superhigh fixes), number of active flights, and total number of flights crossing the designated fix. In case of the sector crossings, the record will have data for sector's capacities, active peaks and total peaks.

Table 10-5 describes the format of the bar chart data file record. Notice that the record has a variant field of *nas_event_t* type.

Table 10-5 Bar Chart Data Record

bar_chart_record			
Library Name: map_lib		**Element Name:** map.ins.pas	
Purpose: To list elements, time intervals, and flights			
Data Item	**Definition**	**Unit/Format**	**Var.Type**
interval_start_time			cal_#timedate_rec_t
element_type		airports, superhi_fixes, hi_fixes, lo_fixes, oceanic, unk, adr	alert_type
elemet_name			char10_t
For airport_departure, airport_arrival:			
arrival_capacities			array of integer
departure_capacities			array of integer
active_arrivals			array of integer
active_departure			array of integer
total_arrivals			array of integer
total_departures			array of integer
For low_fix_crossing, high_fix_crossing, superhigh_fix_crossing:			
low_capacities			array of integer
high_capacities			array of integer
active_low			array of integer
active_high			array of integer
active_superhi			array of integer
total_low			array of integer
total_high			array of integer
total_superhi			array of integer
For sector_crossing:			
total_low			array of integer
total_high			array of integer
total_superhi			array of integer

10.6.4 Weather Data

Weather files are created by the **write weather** command and read back in by the **read weather** command.

10.6.5 The Airway Database

10.6.5.1 The Airway Database — Database File

See Table 10-6 for details on the **Airway Database — Database File**.

Table 10-6. Airway Database File Record

data_record			
Library Name: map_lib		**Element Name:** map.ins.pas	
Purpose: To provide a directory of airways			
Data Item	**Definition**	**Unit/Format**	**Var.Type**
x	x—coordinate in Albers projection		integer
y	y—coordinate in Albers projection		integer
lat	Latitude		real
lon	Longitude		real
code	type of node in airway	VOR, beacon, etc.	integer
name	name of airway		word6

10.6.5.2 Airway Database — Index File

See Table 10-7 for details on the Airway Database Index File Record.

Table 10-7. Airway Database Index File Record

index_record			
Library Name: map_lib		**Element Name:** map.ins.pas	
Purpose: To provide a directory of airways			
Data Item	**Definition**	**Unit/Format**	**Var.Type**
route	name of route		5 chars
color	display color		integer
record_number	record number of start of data in Data Base File	VOR, beacon, etc.	integer

10.6.6 Colors Data

The color file is an ASCII file consisting of several columns. The first column gives the names of the color elements. Words in a color name are separated by a space. For the adjustable colors (for example, red, green, blue, cyan, magenta, and yellow), the remaining columns give the color intensity values of the red, green, blue colors. Each of these color intensities can have an integer value from 0 to 255 inclusive.

For map overlays, prompt background, and window background, the remaining columns specify the color names in the same format as the **colors_default** file.

The color file can be created by the **save colors** command and read by the **restore colors** com-mand. In version 5.0 of the *ASD*, the color file is created in binary format. To provide backward compatibility, the color file in version 5.0 is automatically converted to ASCII format when the file is read through the **RC** command.

The only difference between the **colors_default** file and the color file is that the former does not have any adjustable colors. The adjustable colors are the colors that can be changed by the **AC** command.